UQAC
Université du Québec
à Chicoutimi

MÉMOIRE

PRÉSENTÉ À

L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

TOM CANAC

COMPLETE WEBSITE TESTER: TESTS FONCTIONNELS D'APPLICATION WEB ET COUVERTURE MAXIMALE

JUIN 2018

TABLE DES MATIÈRES

TABLE DES FIGURES

RÉSUMÉ

L'importance prise par les technologies web dans nos vies, l'importance des données qui y transitent, et la qualité désormai attendue dans chaque application et produit utilisé font qu'il est indispensable de tester de façon exhaustive les applications web modernes. Notre piste de travail s'est rapidement orientée vers le développement d'une librairie permettant de tester l'intégralité d'un site ou d'une application web. Lancer la même série de tests sur l'intégralité d'un domaine à été notre objectif final.

Après avoir analysé le besoin de l'industrie, nous nous sommes attachés à faire un état de l'art exhaustif de la littérature scientifique concernant notre domaine de travail. Il s'avère que de nombreuses pistes ont déjà été explorée, sans toutefois aller au bout de la démarche, ou sans livrer d'outil utilisable. Nous avons donc pris le parti d'utiliser ces écrits comme base de travail, en particulier le modèle théorique de l'article de Deyab et Atan (2015). À partir de là, nous avons adapté leur travail à notre interprétation du besoin que nous avons développé plus tôt. Ainsi, nous avons précisé plusieurs concepts, en particulier les connecteurs, mais aussi retiré certains points inadaptés à notre problème, comme le monitoring de serveur.

Nous avons en parallèle développé un outil fonctionnel implémentant notre modèle théorique : Cowtest. Créé en Node.JS, Cowtest permet d'ores et déjà de tester un site complet. À noter

que les tests peuvent être écrits dans n'importe quel langage, avec n'importe quelle librairie de test, sous réserve que la sortie de la librairie puisse être du TAP [1] valide.

1. TAP format contributors (2017)

CHAPITRE 1

INTRODUCTION : DU BESOIN À L'OUTIL COWTEST

Le web est un élément omniprésent dans nos vies quotidiennes. Au premier plan depuis l'arrivée des téléphones intelligents, la plupart de nos interactions avec les ordinateurs, au sens large, se font en ligne. Cette dernière décennie a marqué la modernisation des technologies web. Les sites statiques ont laissé place à des sites dynamiques, en grande partie grâce à la technologie AJAX (Mozilla Developer Network, 2018), pour arriver désormais à des sites non différenciables d'une application native, tenant parfois plus de l'outil que du simple affichage de données.

La modernisation de ces technologies a amené un changement des bonnes pratiques et une complexification de l'environnement de développement. Produire une application robuste de qualité ne laisse pas place au hasard. Les applications web modernes se doivent d'être aussi stables que les applications bureau, tout en faisant face à un environnement diversifié et instable : le navigateur web.

Le domaine du test d'applications web est vaste. Chaque type de test (fonctionnel, unitaire, etc.) demande son propre outil, et dans un domaine en évolution rapide, il est délicat de trouver une solution efficace, durable et interopérable.

1.1 QUESTION DE RECHERCHE

Nous nous concentrerons ici sur les tests fonctionnels réalisés sur la partie « *front* » [1] des applications web. Il existe déjà de nombreuses solutions permettant de réaliser des tests fonctionnels “scénarisés” (une suite d’actions pouvant entraîner des changements d‘état pour vérifier une valeur finale). L’outil le plus connu pour ce faire est Sélénium [2]. Ces outils permettent de vérifier le bon comportement des fonctionnalités d’une application, par exemple l’ajout d’un produit au panier et l’incrément du prix total.

Un autre domaine de test pertinent consiste au lancement d’une série de tests sur l’ensemble des états d’une application. Ces tests permettent par exemple de vérifier la conformité de l’application à certaines normes, que ce soit au niveau de la qualité du code, de l’accessibilité, de la sécurité, ou de tout test envisageable.

Dans les faits, ce type de tests s’avère presque inexistant dans le domaine de l’industrie. Pourtant abordés dans la littérature scientifique (Mesbah et van Deursen, 2009; Thummalapenta et al., 2013; Pellegrino et al., 2015), nous allons nous attacher, dans un premier temps, à analyser la littérature précédente ainsi que le développement des outils disponibles. L’accent sera mis sur l’accessibilité des outils au monde de l’industrie étant donné que ce domaine reste le premier intéressé par les solutions de test.

Comment améliorer la portée et l’automatisation des tests fonctionnels d’applications web sur la totalité d’une application web ?

1. L’interface visible par les utilisateurs finaux
2. `http://docs.seleniumhq.org/`

1.2 DIFFICULTÉS ET VERROUS

1.2.1 CRAWLER

La principale difficulté à lancer un jeu de tests sur une application web complète vient du fait qu'il faut justement récupérer l'ensemble des états de l'application. Cette récupération de l'ensemble des états de l'application se fait par le biais d'un crawler. La difficulté est accrue pour les applications-outil [3] qui ont un caractère moins déterministe que les sites traditionnels.

En effet, dans ce cas, l'architecture de l'application peut dépendre entièrement des données entrées par l'utilisateur, et n'est donc pas toujours déterministe. Si l'on prend l'exemple d'une application de type gestion de projet, permettant de regrouper des tâches dans des projets, des clients, des employés, et des factures. Il est évident que les URL disponibles pour les utilisateurs vont dépendre de leurs propres données. Chaque utilisateur aura des projets aux noms différents, des clients et des employés propres, et une facturation spécifique. Ainsi, les URL vont être générées dynamiquement, en fonction du contenu utilisateur. On ne peut donc pas déterminer à l'avance quelles URL seront valides ou non sur le site.

Dans ce cas, l'idéal serait d'être capable de déterminer quelles variables côté utilisateur fait varier les URL. En effet, un crawler jouant dans l'état de l'art actuel devrait être capable d'analyser les input d'une page, et de trouver les données permettant d'accéder à de nouveaux états de l'application. Pour reprendre l'exemple de gestion de projet sus-cité, un crawler devrait être capable de ne générer qu'un seul projet dans l'application, comprendre que l'URL a été générée par le biais de son entrée de contenu, et ne pas démarrer une boucle infinie de génération de projets aux noms unique, qui seraient vus comme autant de nouvelles pages à découvrir. Dans les faits, ce point est très rarement pris en compte.

3. Offrant une fonctionnalité et non pas un contenu

Le cas particulier de la connexion utilisateur résiste en particulier aux crawlers modernes, pour une raison évidente, puisque passer cette barrière de façon automatisée reviendrait à pirater le site.

1.2.2 COMMUNICATION AVEC UNE APPLICATION AJAX

Il est notable qu'aucun standard de crawl et de description du cheminement dans une application AJAX n'existe actuellement. En effet, il est simple d'enregistrer un changement d'URL, mais il est notablement plus complexe d'enregistrer et de décrire quelle interaction avec la page à déclenché quelle modification. Il va donc être à notre charge de définir notre propre façon de décrire le chemin parcouru dans l'application, les interactions avec le DOM, ainsi que les entrées réalisées dans les éléments modifiables.

1.2.3 PLURALITÉ DES LOGIQUES MÉTIER

Laisser le champ libre aux différentes logiques métier est également un point délicat. En effet, chaque milieu professionnel peut avoir des exigences différentes en matière de test d'application. Du test d'interface aux tests fonctionnels, en passant par du temps de réponse, il paraît logique que l'outil développé puisse être adapté à des besoins si différents.

1.3 OBJECTIFS

Nous avons établi plusieurs objectifs afin de mener notre recherche à bien :

1. Réfléchir à une façon efficace et appropriée de tester un site complet ;
2. Déterminer un modèle théorique et une architecture fiable ;

3. Développer une implémentation fonctionnelle respectant au plus près notre modèle théorique.

Ces trois objectifs principaux seront réévalués en fin de mémoire, afin de déterminer si oui ou non, nous avons réussi à respecter notre postulat et notre orientation de départ.

1.4 PLAN

1.4.1 ÉTAT DE L'ART

Dans un premier temps, nous allons nous concentrer sur analyser l'avancement actuel de la littérature scientifique sur notre sujet de recherche. Nous travailleron sur les articles scientifiques publiés lors des 6 dernières années, comportant les mots clefs **crawl**, **test** et **web**, et traitant de l'utilisation d'un Crawler dans l'objectif de lancer une série de tests. Après avoir exposé nos concepts clefs, nous ferons notre revue de littérature des articles pertinents, suivie d'une analyse et d'une discussion sur la place de notre travail en parallèle à l'existant.

1.4.2 MODÈLE THÉORIQUE

Ensuite, nous présenterons le modèle théorique que nous avons établi à la suite de l'état de l'art. Le modèle théorique représente à la fois notre solution intellectuelle préférée, mais aussi la fondation de notre implémentation, qui sera malgré tout, comme nous le verrons, plus réaliste. Le modèle théorique original vise à permettre une bonne orchestration des concepts qui seront présenté plus loin. L'objectif étant un travail commun optimisé et efficace de toutes les briques logicielles.

1.4.3 IMPLÉMENTATION

Finalement nous présenterons l'implémentation logicielle de notre modèle. De la conception à l'utilisation de l'outil, en passant par nos choix technologiques.

Première partie

État de l’art

CHAPITRE 2

CONCEPTS

2.1 TECHNOLOGIES WEB

2.1.1 APPLICATION WEB

Le terme « application web » est large, ici, il définira l'ensemble des sites consultables par le biais d'une URL. Cela comprend donc :

1. les sites statiques (type vitrine, de présentation de données) ;
2. les sites dynamiques (forums, blogs, etc.) ;
3. les applications mobiles construites sur des technologies web, souvent affichées dans des webviews (applications Instagram, Facebook, etc.).

Toutes ces applications sont servies, la plupart du temps, par le biais du protocole HTTP [1], et visionnables dans un navigateur web. Le navigateur web interprète les données servies par le serveur HTTP, composées de HTML, CSS, et JavaScript. D'autres technologies peuvent entrer en jeu, mais ces trois constituent la base de la majorité des applications web. À noter que la partie côté serveur comporte de nombreuses autres technologies, qui, comme nous le verrons plus tard, ne nous concernent pas directement dans notre étude.

1. HyperText Transfer Protocol, protocole défini par World Wide Web Consortium (2015)

HTML Langage de balises, HTML est purement descriptif. Spécifié par le World Wide Web Consortium (2017b), il permet de mettre de la structure dans le document, organiser et hiérarchiser le contenu, voire dans ses dernières versions, ajouter une certaine sémantique à la page web. Le langage HTML se veut le squelette de toute application web, et ne comporte aucune logique métier.

Listing 2.1 – Exemple de code HTML simple

```
<!DOCTYPE HTML>
<HTML>
<head>
  <title></title>
</head>
<body>
        <p class=''une-classe''></p>
        <div id=''un-id''></div>
</body>
</html>
```

Dans l'exemple ci-dessus, nous voyons la structure de base de n'importe quelle page valide HTML5. La page démarre par le "doctype" HTML, c'est l'instruction qui va déterminer quelle version de HTML est utilisée par le site. S'en vient ensuite la balise "head", qui permet de donner un certain nombre d'informations à propos de la page, en particulier le titre, mais aussi de lier des feuilles de style CSS par exemple. La partie principale de toute page HTML est le "body", c'est dans cette balise que va résider le contenu de la page finale. En l'occurrence nous avons deux tags contenus dans le "body" : un "p" qui est un paragraphe, avec en l'occurrence une classe "une-classe", est une div (qui peut être imaginée comme une boîte), spécifiée par un ID "un-id". La classe et l'ID vont servir principalement pour le CSS, comme nous allons le

voir dans le paragraphe suivant.

CSS langage déclaratif statique spécifié par le World Wide Web Consortium (2017a), CSS se concentre purement sur la partie esthétique de l'application. Il permet, grâce à un ensemble de sélecteurs et de méthodes de ciblage des éléments de la page HTML, de mettre en forme la typographie, les alignements, les couleurs, et bien d'autres éléments, plus ou moins complexes.

Listing 2.2 – Exemple de code CSS simple

```
body {
        background-color:red;
        color:yellow;
}
.une-classe {
        height: 300px;
        width:250px;
        border-size: 1px;
        border-color: green;
}
#un-id {
        font-family: Helvetica, Arial, sans-serif;
}
```

Le code CSS ci-dessus comporte trois sélecteurs distincts : body, .une-classe et #un-id. Le premier, body, applique deux propriétés de style à l'élément HTML body : une couleur de fond, et une couleur de typographie. Le second, .une-classe, applique des propriétés de taille (height et width), ainsi qu'une bordure verte de 1 pixel de large, à tous les éléments HTML comportant la classe "une-classe" (les sélecteurs de classes commencent tous avec un point). Ces styles

seront donc répétés sur plusieurs éléments de la page. Le dernier sélecteur, démarrant avec un croisillon, vise un id HTML. La propriété CSS de cet ID définit un style de typographie : le navigateur cherchera à activer la police Helvetica dans un premier temps, suivi par Arial (avec une priorité moins haute), pour enfin finir sur n'importe quelle police sans empattement en dernier recours. Le sélecteur de type ID indique que ce style ne sera appliqué qu'à un seul élément de la page, étant donné que les ID HTML se doivent d'être uniques au sein d'une seule et même page HTML.

JavaScript Pierre angulaire du web moderne, JavaScript est désormais un incontournable de toute application web. Spécifié par l'organisme ECMA International (2017), il comporte l'ensemble de la logique côté client de la page. Associé au HTML sous forme de script, le code JavaScript va s'exécuter une fois téléchargé dans le navigateur web, de façon transparente pendant le chargement de la page. Il est capable de modifier le code HTML de la page, réaliser des calculs, faire des requêtes au serveur, à des pages, et bien d'autres opérations avancées.

Listing 2.3 – Exemple de code JavaScript

```
const tmp = [1,2,3,4,5];


const incrementedArray = tmp.map((num) => {
  return num++;
});


console.log(incrementedArray);
//[2,3,4,5,6]
```

L'exemple ci-dessus montre la syntaxe du JavaScript au travers d'un code simple. Nous initions une variable contenant un tableau (on note l'absence de délacration de type), puis

nous utilisons une fonction de programmation fonctionnelle (map) pour créer une copie du tableau, incrémentée. Enfin, le résultat est loggué dans la console.

Traditionnellement, la navigation dans une application web se fait grâce à des liens hypertextes (couramment appelés “hyperliens”, ou simplement “liens”). Pourtant, à l’heure actuelle, il est délicat de parler de « *pages* » dans le domaine des applications web. En effet, certaines applications fonctionnent sans nécessairement modifier l’URL de l’application, en particulier grâce à la technologie AJAX, qui permet d’exécuter des requêtes externes à partir de JavaScript. Nous parlerons ici donc « *d’état* » de l’application. Pour les sites statiques classiques, cela correspond aux pages, accessibles par le biais de liens, et pour une application moderne, cela s’apparentera aux différents écrans accessibles par le biais d’actions utilisateurs.

Par souci de simplicité, nous appellerons ici les applications statiques le « web 1.0 », composé de pages HTML simples, les applications utilisant AJAX, le « web 2.0 », et les applications utilisant des technologies JavaScript avancées de type React, Angular, Ember et consorts, le « web moderne ».

2.1.2 FRONT ET BACK END

Une application web est définie dans deux contextes principaux : le front-end, côté client, et le back-end, côté serveur. Ces deux contextes fonctionnent main dans la main pour livrer une application complète à l’utilisateur.

Le back-end est l’ensemble du code source exécuté côté serveur. Cela comprend, par exemple, le code php d’un site, ou encore du code node.js pour une application dynamique. C’est ici qu’une grande partie de la logique est souvent comprise, et que l’application est générée, avant d’être envoyée au client par le biais du protocole HTTP.

Le front-end, à l'opposé, est l'ensemble du code exécuté côté client : dans le navigateur de l'internaute. Cela comprend tout le HTML, CSS et JavaScript de l'application. Ces trois technologies représentent la grande majorité des technologies front-end modernes.

Il est important de garder en mémoire que l'utilisateur final d'un site ou d'une application web peut très simplement altérer le DOM d'une page, grâce à l'inspecteur d'éléments de son navigateur. Il est donc important en tant que développeur d'une application web de ne pas faire confiance aux données du DOM, et donc de ne pas mettre de sécurité côté client.

2.1.3 AJAX

L'Asynchronous JavaScript And XML (AJAX) est une technologie au coeur du JavaScript frontend. Utilisé sous forme de requêtes, AJAX permet de modifier le contenu d'une page au fil de la consultation. L'URL de la page consultée n'a donc plus besoin de changer pour avoir un contenu différent. Ce point peut sembler anodin, mais c'est en réalité le plus gros changement de paradigme du web des 10 dernières années.

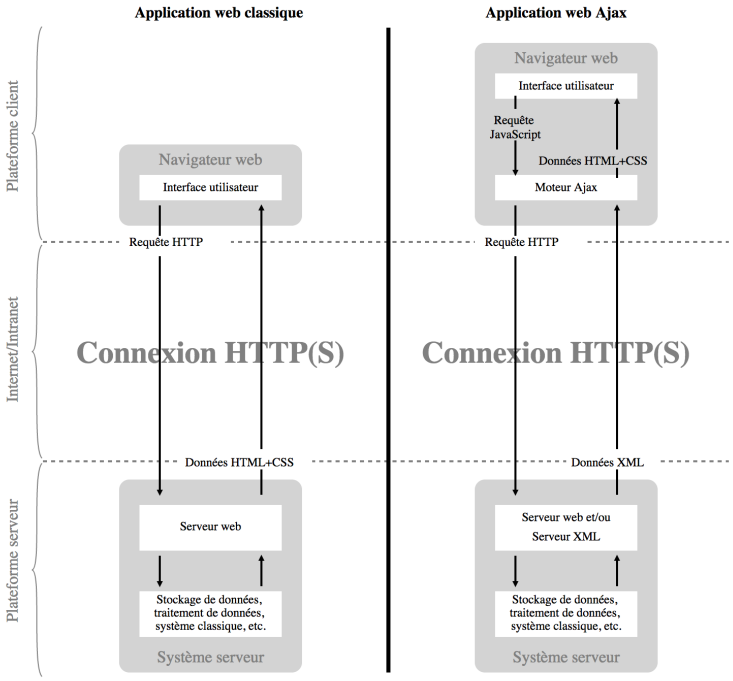
La figure 2.1 illustre bien le changement de fonctionnement entre une requête classique et une requête AJAX.

Dans une application web traditionnelle, le navigateur lance une requête HTTP (ou HTTPS si le site utilise une surcouche de sécurité). Le serveur traite cette requête, génère le contenu de la réponse, et renvoie cette dernière, a priori composée de HTML, CSS, et média (images, musiques, etc.). Le navigateur va, dès réception de la requête, rafraîchir l'écran de l'utilisateur, changer l'URL, et charger la nouvelle page.

Dans une application web AJAX, la requête se fait toujours via HTTP(s), mais les données sont différentes. En effet, le serveur, connaissant la nature de la requête, retourne des données

structurées, sous forme de XML, ou encre de JSON. Le moteur AJAX va interpréter la réponse, ce qui permet d'éviter le rechargement complet de la page et mettre à jour directement le DOM.

Figure 2.1 – Fonctionnement d'une requête AJAX. Source : Wikipédia, Article AJAX.



2.1.4 DOM

Le Document Object Model est une structure de données contenue dans le navigateur web formé à partir du code HTML d'une page. Il s'en distingue toutefois par l'interprétation du JavaScript. En effet, le DOM contient toutes les modifications qu'on pu être apportées au HTML par le code JavaScript. Cette différence est importante, car cela met en avant le fait que le DOM n'est pas immutable. En effet, il peut varier et être altéré pendant la consultation de la page par diverses méthodes JavaScript (l'AJAX en particulier).

Le DOM étant le HTML combiné au JavaScript interprété par le navigateur, il est important de

noter qu'une même page fraîchement chargée peut avec un rendu de DOM différent selon les navigateurs. En effet, certains moteurs de rendu ont des exceptions, des comportements non standards, ou même parfois des bugs d'interprétation spécifiques. Autant de variations qui font que le DOM peut véritablement être considéré comme le "rendu" du site dans le navigateur.

Ainsi, une page comportant le code HTML suivant :

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
  <script>
        document.getElementById('un-id').classList.add('une-classe')
  </script>
</head>
<body>
        <div id=''un-id''></div>
</body>
</html>
```

Une fois chargée, et le JavaScript interprété, aura un DOM correspondant au code suivant :

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
  <script>
        document.getElementById('un-id').classList.add('une-classe')
  </script>
```

```
</head>
<body>
        <div id=''un-id'' class=''une-classe''></div>
</body>
</html>
```

En effet, le code JavaScript contenu entre les balises script vise l'élément à l'ID "un-id", et lui ajoute la classe "une-classe". La modification du HTML est réalisée, encore une fois, côté client, purement dans le navigateur. Le serveur n'a jamais connaissance d'une div contenant à la fois l'id "un-id" et la classe "une-classe".

Le DOM est soumis à une spécification rédigée et administrée par le World Wide Web Consortium (2004). La dernière version valide date de 2004.

2.1.5 CRAWLER

Un crawleur web est un logiciel chargé de naviguer sur le web de façon autonome. Traditionnellement, ces logiciels sont utilisés pour récupérer de l'information sur le web entier afin d'alimenter des moteurs de recherche. Aujourd'hui, on constate que les crawleurs peuvent servir dans un grand nombre de domaines (classification de fichiers, recherche d'information spécifique, recherche d'informations généraliste, etc.).

En théorie, la plupart des crawlers classiques ont un fonctionnement similaire : À partir d'une URL de base (généralement appelée "seed"), le logiciel scanne la page à la recherche d'hyperliens, conservent ceux qui correspondent à un filtre défini par l'utilisateur [2], puis relance son scan de façon récursive sur les liens sélectionnés, s'ils n'ont pas déjà été explorés. Cela

2. Les liens internes uniquement, les liens vers des images, les liens contenus dans un sélecteur CSS particulier, etc.

permet de récupérer de façon optimisée et construite l'ensemble des pages d'une application web statique.

Cela dit, à partir du web 2.0, on constate une difficulté grandissante à crawler les applications web. En effet, l'arrivée d'AJAX à entraîné un changement dans les systèmes de navigation : les pages, identifiées historiquement par leurs URL sont devenues des états, identifiables par la valeur du DOM.

Ce changement d'identification entraîne un nombre conséquent de changements sur le crawl d'application web moderne, comme nous allons le voir tout au long de ce travail.

2.1.6 ORCHESTRATION

L'orchestration de processus est le fait de faire travailler ensemble tous les processus d'un service. L'optimisation des processus, au sens informatique du terme, n'est qu'une partie du domaine de l'orchestration. En effet, il s'agit ici d'optimiser l'ensemble des processus métiers de l'application, de les faire fonctionner en synergie, du mieux possible, afin de créer un cadre de travail efficace et optimisé pour l'utilisateur final. De nombreux critères peuvent être pris en compte dans l'orchestration, par exemple la performance des opérations, la consommation d'énergie, etc.

L'orchestration permet de :

1. Réduire les délais de traitement ;
2. Rendre fiable l'exécution d'un grand nombre de processus en parallèle ;
3. Gérer l'utilisation des ressources de la machine hôte.

Dans notre cas, l'orchestration sera exclusivement liée à la gestion du lancement des tests. En effet, le fait que chaque test doive être lancé pour chaque page du site se traduit par un grand

nombre de sous-processus. Un grand nombre de processus implique une grande responsabilité vis-à-vis de l'utilisation des ressources de la machine du client.

2.2 TESTING

2.2.1 BUG

Avant de travailler sur le test d'un logiciel, sa validation ou son invalidation, il est important de définir dans quel cas notre test est valide, ou invalide. Cela passe par la définition de la notion de bug. Au sens large, un bug est un comportement non désiré. Si on cherche une description plus formelle, la notion de condition est utile : selon l'IEEE, c'est la différence entre les "conditions existantes et les conditions requises" d'un composant logiciel (IEEE, 1990).

Dans notre cas, comme nous allons le voir par la suite, c'est l'utilisateur qui déterminera manuellement les conditions de validation des tests. Soit par une approche positive (ceci est juste) ou par une approche négative (cela ne doit pas arriver). Par exemple, un utilisateur pourrait chercher à détecter les bugs suivants :

1. L'absence d'un élément sur la page. Cela pourrait être d'un point de vue code HTML, mais aussi de contenu textuel, d'image, de code HTTP, un copyright, etc. ;
2. L'invalidité d'une page aux normes W3C ;
3. La présence de code d'erreur ou d'alerte dans la console du navigateur.

À noter que les bugs détectables peuvent également être très spécifiques à l'utilisateur, et donc dépasser la notion de bug informatique simple :

1. Le respect d'une charte graphique (présence ou absence de couleurs) ;

2. Accessibilité du code HTML de la page pour les utilisateurs non voyants ;
3. Respect de normes de développement internes à l'entreprise.

2.2.2 TEST

On peut distinguer deux grands types de tests logiciels : « *white box* » et « *black box* ». Le premier se fait en ayant accès au code source de l'application. Le second quant à lui se fait sans accès à la source, uniquement à l'interface « *front* » finale, manipulée par l'utilisateur.

Cette recherche se concentre sur les tests fonctionnels réalisés sur les tests de type boîte noire, ciblés sur l'interface visible par les utilisateurs des applications web. L'objectif étant de pouvoir tester le produit final utilisable.

Dans ces tests boîte noire, nous retiendrons un certain nombre de spécialités de tests :

1. Manuels : par le développeur, le testeur, une agence spécialisée... ;
2. Fonctionnels : tests de scénarios utilisateurs et de fonctionnalités du site (suite d'actions et vérification de résultats) ;
3. Conformité : le produit suit-il des standards définis ?
4. Montée en charge : le produit tient-il la montée en charge ?
5. Régression : la mise à jour du produit n'a-t-elle pas impacté l'existant ?

Le test est un domaine de recherche travaillé, étudié, et théorisé. 7 grands principes sont généralement retenus pour décrire le monde du test logiciel :

1. Un test sert à détecter un défaut logiciel
2. Il est impossible de tester exhaustivement une application

3. Afin d'optimiser les coûts et l'efficacité, il faut tester tôt
4. Les bugs appellent les bugs. Il faut concentrer l'effort de test sur les parties connues comme étant moins stables
5. Le paradoxe des antibiotiques : si on teste toujours les mêmes choses, les réparations apportées au logiciel vont être réalisées de façon à passer le test. Pas à résoudre le bug. On doit renouveler et diversifier la série de tests.
6. Le type de test dépend de la nature du contexte et de l'objet à tester. Une application web dynamique n'est pas testée de la même façon qu'une carte de détonateur d'explosif de mine.
7. L'absence d'erreur d'un système signifie que ce qui a été testé est correct. Cela ne signifie pas que le logiciel sera un succès ou répond de façon pertinente au besoin de base.

Afin de fonctionner pleinement, une suite de test repose sur quelques concepts clefs, en particulier :

1. Oracle : c'est l'élément logiciel qui va valider ou invalider les conditions de passage du test. C'est l'oracle qui détient la vérité sur l'équivalence de deux valeurs, sur la justesse d'une variable, ou encore sur un ensemble de données à vérifier.
2. Verdict : un test a deux valeurs possibles, comme un booléen : vrai ou faux. Un test passe, ou ne passe pas. Il est tout à fait possible d'accompagner l'échec d'un test d'informations additionnelles pour permettre à l'utilisateur de mieux comprendre la situation, mais ultimement, le verdict est simplement un 0 en cas de réussite (absence de données remontées), ou un 1 en cas d'échec (explications sur la cause de l'échec).

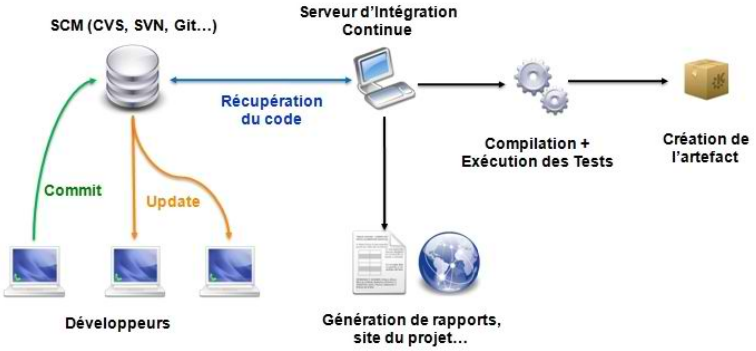
2.2.3 INTÉGRATION CONTINUE

L'Intégration continue (IC) est un terme regroupant toutes les pratiques permettant de vérifier qu'une mise à jour du code d'une application ne crée aucun effet de bord ou aucune régression par rapport à la version existante de l'application, et ce de façon automatique.

Un workflow d'intégration continue permet de mettre en place des automatismes assez avancés, par exemple une réinstallation complète de l'outil à chaque itération des tests, ou encore la remise à zéro d'une base de données. De façon simplifiée, une étape d'intégration continue est une suite d'opérations scriptées par le testeur. Si le script retourne le code 0, l'intégration est réussie, si le code n'est pas 0, c'est donc un code erreur (selon les standards UNIX), et le processus d'intégration continue interrompra le déploiement.

Comme nous le voyons dans la figure 2.2, une fois complété, le processus d'IC livre un "artéfact", qui est le résultat testé et compilé de l'application finale. De façon générale, cet artefact peut être sous n'importe quelle forme : exécutable, archive, etc. Dans le cas d'une application web, la nature de l'artefact peut varier, mais nous pouvons affirmer que, de façon générale, il sera constitué d'un ensemble de fichiers CSS, HTML, JavaScript et ressources graphiques.

Figure 2.2 – Modèle d'utilisation de l'intégration continue. Source : Sébastien (2013)

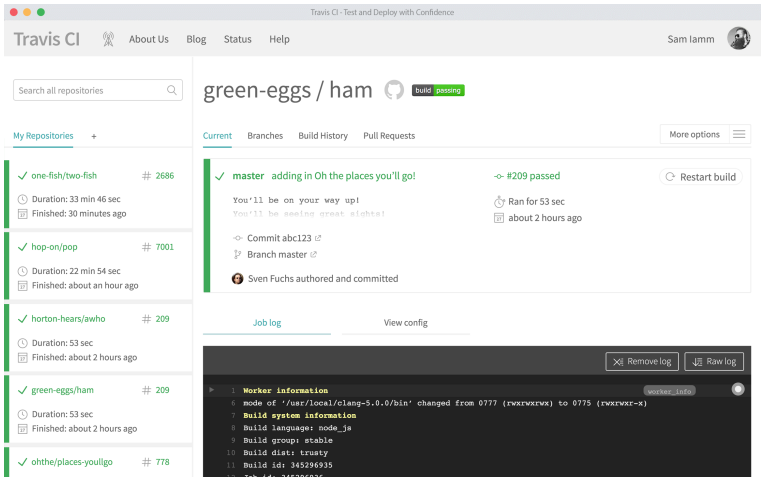


L'objectif de l'intégration continue est d'accélérer le temps de développement en limitant l'apparition de bugs et de comportements inattendus. Selon Sébastien (2013), l'IC permet de gagner en productivité, en efficacité, mais aussi en qualité de code.

Un outil désirant être compatible avec un workflow d'IC devrait offrir un moyen de connexion à au moins un serveur d'IC. Cela permettrait à l'utilisateur final d'ajouter l'outil en question à un processus déjà existant, et réduirait donc l'appréhension liée à l'adoption d'un nouvel outil.

Actuellement, l'outil majeur du marché de l'intégration continue est Travis CI. Il tire sa popularité de son intégration gratuite à tous les projets open source de l'héberger de projet Github. Comme nous le voyons dans l'image suivante, l'interface, extrêmement simple, permet de valider chaque commit sous une batterie de tests. Comme nous le voyons dans l'impression écran, il y a très peu de contrôles dans l'interface : toutes les tâches sont automatisées. En cas d'échec, le commit est signalé en rouge, et il est à la charge de l'utilisateur de modifier son code afin de passer le test échoué.

Figure 2.3 – L'interface graphique de Travis CI, leader du marché de l'intégration continue



2.3 GÉNÉRALITÉS

2.3.1 ARCHITECTURE MODULAIRE

Le concept d'architecture modulaire mis en avant est somme toute proche de la programmation orientée objet. Chaque partie du logiciel est dans un premier temps conceptualisée et programmé de façon indépendante, avec une interface de communication définie en amont et stable. Ces parties, qui sont nos modules, sont capables d'être utilisées par n'importe quel autre script qui prendrait le temps de respecter l'interface publique du module. Les modules sont ensuite utilisés dans un script principal, ou bien un autre module, qui aurait alors simplement une dépendance vers le premier. Comme nous le verrons plus tard, certains langages permettent aux programmeurs de bénéficier très simplement d'une publication et d'un versionnement de leurs modules.

CHAPITRE 3

LITTÉRATURE

Tout état de l'art répond à un ensemble de critères de sélection des articles. En l'occurrence, nous avons sélectionné les articles contenant l'ensemble des mots clefs suivants : **crawl**, **test** et **web**. Les articles sélectionnés doivent traiter de l'utilisation couplée d'un crawleur pour effectuer une série de tests.

La limite temporelle est située à 6 ans d'ancienneté maximum, soit 2012. Cela permet d'avoir un jeu d'articles à jour techniquement, tout en remontant assez loin pour comprendre les enjeux actuels du domaine.

Les articles seront analysés selon les critères suivants :

1. Implémentation : est-ce que les auteurs de l'article fournissent un support logiciel pour utiliser les idées proposées dans leurs articles ? Cette solution est-utilisable en l'état ?
2. Personnalisation : la solution permet au testeur d'écrire le code de tests entièrement personnalisés ;
3. Intégration continue : la solution est compatible avec des outils d'intégration continue (Travis, Jenkins, etc.). L'intégration continue étant un standard dans l'industrie depuis maintenant plusieurs années, nous considérons ici qu'une solution ayant pour objectif

de s'intégrer au monde de la production d'applications web devrait être à jour sur ce point.

3.1 WEBMATE

L'article de Dallmeier et al. (2012) part du constat qu'il est long et délicat de maintenir une suite de test, ce serait même la raison principale pour laquelle relativement peu de compagnies testent leurs solutions extensivement. L'objectif annoncé de WebMate est de réparer cet état de fait, de rendre le test automatique d'applications web 2.0 efficace et simple.

L'application contrôle un navigateur sans tête : PhantomJS, basé sur l'interpréteur WebKit. WebMate base son identification des états d'application sur le contenu du DOM, non pas sur les URL, et permet donc de naviguer dans des applications type web 2.0.

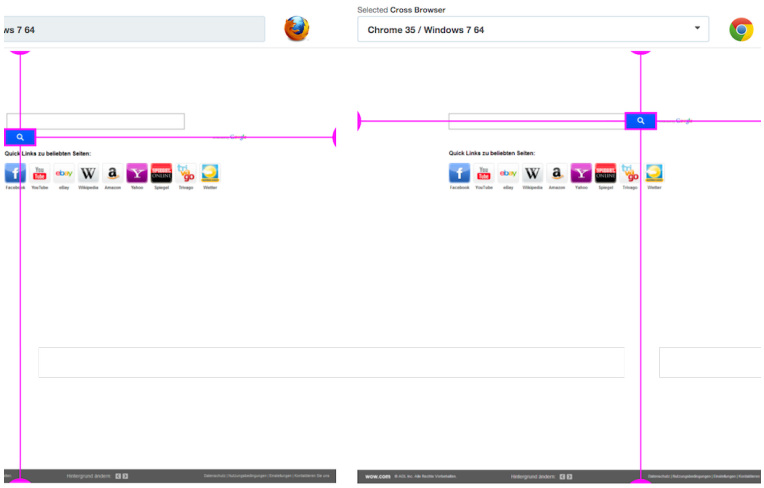
Afin de détecter les éléments qui sont susceptibles de changer l'état du DOM, WebMate se base sur les fonctionnalités des librairies jQuery et Prototype. WebMate implémente une couche d'abstraction sur le DOM, afin d'optimiser le parcours de l'application, tout en détectant les différentes actions qui pourraient amener aux mêmes états de l'application, afin d'optimiser la taille du graphe de parcours.

Grâce à PhantomJS, WebMate permet de réaliser de tests « *Cross Browser* » [1]. Pour ce faire, WebMate utilise une approche comparative : un navigateur est défini comme base fonctionnelle, et tous les comportements changeants de cette base valide seront considérés comme anormaux.

Suite à la publication de leur article scientifique, le projet WebMate est passé de la recherche à l'industrie avec le projet WebMate.io. Leur solution a été en production sous forme de

1. Tester l'uniformité du site sous différents navigateurs.

Figure 3.1 – Webmate, une solution désormais spécialisée dans les tests d'interface.



SAAS [2]. On constate que leur logiciel s'est concentré test d'interface, et à abandonné la possibilité des tests personnalisés. Comme nous le voyons dans la figure 3.1, WebMate a pris une orientation un peu différente, se spécialisant dans la comparaison d'impressions écrans prises avec différents navigateurs. WebMate met ensuite en valeur les différences visuelles entre les images récupérées, afin de permettre à l'utilisateur d'avoir une meilleure vision des potentiels problèmes d'affichages de son application web.

Hélas, depuis le début de l'écriture de notre travail, le projet Webmate est tombé à l'eau. En effet, plus aucune trace du projet n'est disponible en ligne, et leur nom de domaine a été libéré. La direction prise par le projet ne leur a probablement pas permis de perdurer dans le domaine professionnel commercial.

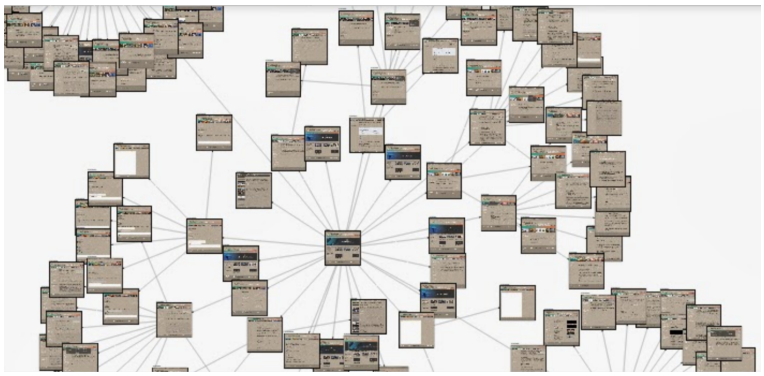
2. Software As A Service

3.2 CRAWLJAX

Le constat de base de l'article de Mesbah et al. (2012) est que les applications AJAX sont difficiles à tester. Leur objectif annoncé est de rendre possible le test automatisé d'applications AJAX. Pour ce faire, ils se basent sur leur outil présenté dans leurs publications précédentes Mesbah et al. (2008) Mesbah et van Deursen (2009) : Crawljax. Crawljax est un crawleur spécialisé dans l'analyse d'applications AJAX.

Crawljax permet de crawler un site complet et de réaliser un « *web State Machine* », c'est-à-dire une représentation sous forme de graphe du crawl réalisé. L'outil permet également de visualiser ce State Machine, comme illustré dans la figure 3.2, où nous voyons le graphe d'exploration interactif.

Figure 3.2 – Extrait de l'interface de Crawljax : le compte rendu visuel d'un crawl.



En interne, les liens entre les nœuds du graphe sont représentés par une combinaison d'élément du DOM via leur sélecteur XPATH et d'une action utilisateur. L'outil possède une liste d'éléments suspectés d'être cliquables grâce à AJAX (a, div, input, img, accompagnés par les personnalisations de l'utilisateur). Les actions les plus courantes (click, hover, double click, etc.) sont lancées sur ces éléments, avec une vérification de changement d'état du DOM. L'article insiste sur de nombreux détails d'implémentation technique du crawleur, confirmant

leur volonté de créer un logiciel de référence dans le domaine.

L'auteur se penche sur l'analyse de site par « invariant ». Un invariant est une valeur censée valoir « *true* » durant toute l'exécution du programme. En langage de test, c'est comparable à une assertion. L'analyse d'états AJAX par invariant revient donc à exécuter des tests et à vérifier leur valeur retour.

Quelques exemples de tests applicables au DOM sont donnés : validation du DOM, non-présence de messages d'erreur (500, 404), accessibilité, etc. L'article décrit également la possibilité d'écrire ses propres invariants en conditions XPath, Java ou JavaScript. Il est également possible d'effectuer des tests sur la formation de l'URL, ou de vérifier la visibilité d'éléments du DOM.

Il est également possible d'appliquer des invariants au « *web State Machine* », par exemple : vérifier l'absence de liens morts (code 404), la consistance du bouton retour (en particulier dans les applications AJAX), etc. Ici aussi, il est possible d'ajouter des tests personnalisés.

Le logiciel est très ouvert et permet de vraiment personnaliser le comportement du crawleur et des tests via un système de plug-ins complet. L'intégration continue semble avoir été développée, mais la page n'est plus disponible en ligne, ce qui ne nous permettra pas de juger du développement de cette fonctionnalité à l'heure actuelle.

3.3 WEBMOLE

L'article de Le Breton et al. (2013) présente leur outil Webmole. Webmole est un crawler d'applications AJAX fonctionnant dans un navigateur web.

Webmole utilise un système « d'Oracles » pour évaluer des conditions écrites en JavaScript

par l'utilisateur. Les Oracles sont des fonctions prenant en argument un état du DOM, et retournant vrai ou faux. Les Oracles permettant de manipuler l'exploration du robot de façon précise et personnalisée.

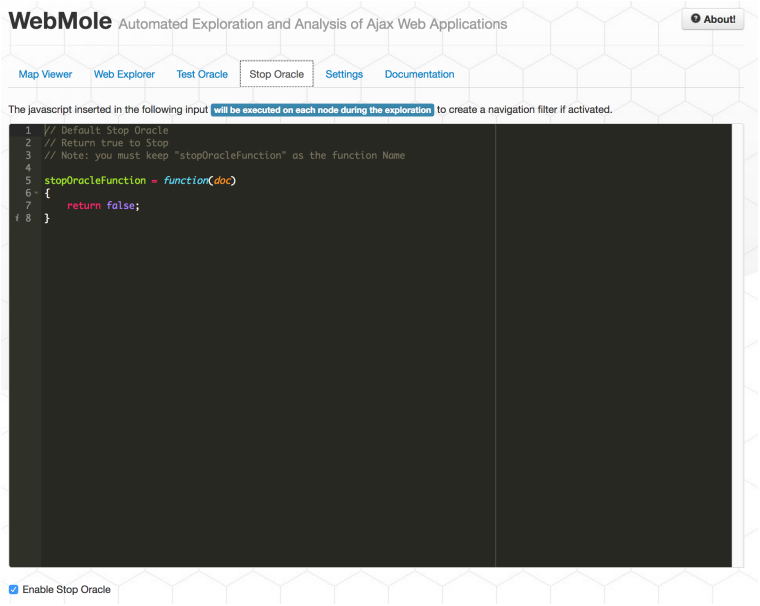
Webmole est un outil open source au code disponible depuis la publication de l'article. L'installation se fait au moyen de machine virtuelle. Cela demande quelques connaissances, mais tout développeur PHP devrait pouvoir en venir à bout. Les Oracles sont entièrement personnalisables et permettent même de contrôler l'évolution du crawleur dans le site. L'intégration continue n'est pas disponible et ne semble pas être une direction retenue pour le développement de l'application.

Au lancement, Webmole propose deux modes d'exploration : manuel et automatique. L'exploration manuelle permettra à l'utilisateur d'aller cibler les pages désirées de façon précise, mais relativement lente (vitesse de navigation humaine). Le mode automatique réalisera un crawl complet de l'application ciblée. Avant de lancer l'exploration, l'utilisateur peut définir des conditions en JavaScript, à l'aide d'un éditeur embarqué, afin de marquer certaines pages, ou encore stopper l'exploration.

L'application parcourt ensuite le site visé en direct, et affiche les résultats à l'écran, ainsi qu'une légende du code couleur utilisé, comme nous le voyons dans la figure suivante.

À noter que les conditions JavaScript, ici appelées Oracles, sont là afin de permettre à l'utilisateur de créer un filtre de navigation. Leur but est de guider le crawler dans son exploration du site, non pas de créer une série de tests à rouler sur le site (bien qu'a priori, cela puisse indirectement être réalisable).

Figure 3.3 – L'éditeur et la syntaxe des Oracles de Webmole



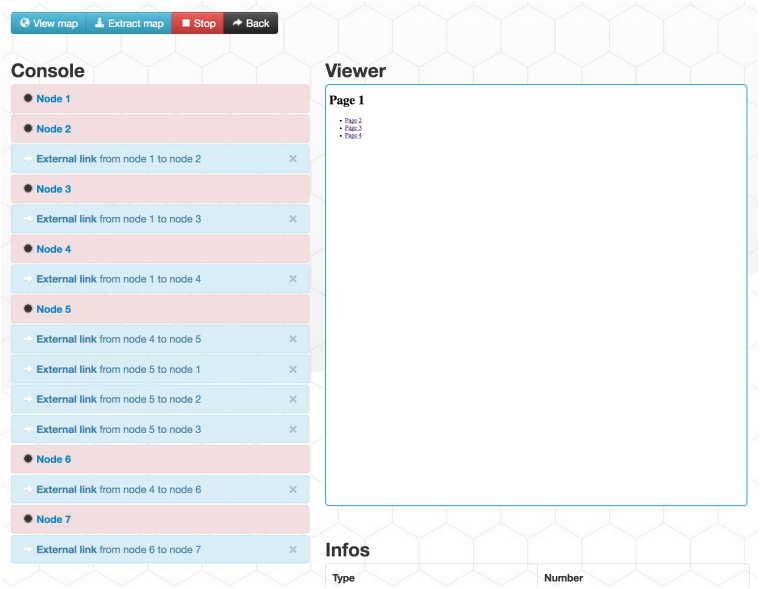
3.4 REGRESSION TESTING

L'article Raina et Agarwal (2013) met en oeuvre une suite logicielle spécifiquement créée pour conduire des tests de régression. L'outil intègre un crawler web 1.0, donc sans support des applications AJAX. Il compare ensuite le contenu du DOM de chaque crawl, pour mettre en avant les modifications du contenu de la page.

Les auteurs mettent à l'épreuve leur outil sur le CMS Wordpress, et font remonter avec succès les modifications de page détectées.

Les auteurs ne donnent aucun accès à leur code source ou bien à une version téléchargeable de l'application. Il nous est donc impossible de détailler le fonctionnement de l'outil développé. A priori, cette solution ne permet pas la personnalisation de tests, seulement la comparaison entre deux pages HTML. D'ailleurs, il est notable que l'outil ne mette pas en avant les bugs, ou ne réponde à aucune réelle assertion, mais permette juste de détecter les changements HTML

Figure 3.4 – Le résultat du crawl d'une application par Webmole



d'une page. Cela limite grandement l'utilisation de l'outil dans le cadre d'un processus de test personnalisé et adapté à une logique métier précise.

3.5 AUTOMATED SYSTEM TESTING

L'article de Tanida et al. (2013) se base sur l'outil déjà présenté Crawljax, qu'il améliore du point de vue des tests de conformité de modèles. L'outil permet de valider des contraintes de navigation sur le « *State Transition Graph* » [3] obtenu par le crawleur.

L'objectif principal est donc de valider les séquences de navigation, les liens, entre les états d'une application web 2.0.

Par exemple, l'outil pourrait permettre de vérifier que toutes les pages sont accessibles à partir d'interactions, ou bien que l'accueil soit accessible depuis tous les écrans. Une proposition de langage de description de séquences de navigation est faite dans l'article.

3. Synonyme de « *web State Machine* »

L'outil proposé tient en deux temps : une extension Crawljax nommée « GuidedCrawl » et un validateur de modèle de navigation. L'extension permet ajouter à Crawljax la possibilité d'avoir des suites d'actions définies par l'utilisateur qui prendront le dessus sur le crawl automatique afin de pouvoir valider des comportements spécifiques propres à certaines applications. Le second outil, Goliath, quant à lui, est spécialisé dans la validation des données sorties par Crawljax au regard de modèles définis par l'utilisateur.

Des exemples d'utilisation de leur application sont ensuite proposés, afin de valider leur approche.

L'implémentation du modèle proposé est faite en deux temps : une amélioration de l'outil existant Crawljax par le biais d'un plug-in, et la livraison d'un second outil : Goliath, l'outil de vérification de conformité. Leur outil bénéficie donc d'un environnement stable et éprouvé (Crawljax), ce qui permet de ne pas réinventer la roue et de bénéficier des avancées de ce premier. À noter que leur champ de test est spécialisé et ne rend pas Crawljax plus ouvert, mais lui permet d'exceller dans un domaine précis : la vérification de contraintes de navigation. L'outil ne correspond donc pas totalement à nos critères d'analyse.

3.6 LEVERAGING EXISTING TESTS

L'article de Milani Fard et al. (2014) propose une approche différente de tous les autres papiers ici retenus. En effet, partant du principe que les tests dits "scénarisés" [4] sont de bonne qualité et démontrent une connaissance réelle de l'application testée, les auteurs proposent une méthode pour récupérer cette connaissance de l'information afin de la coupler à la puissance d'un crawler. Le crawleur est amélioré par les informations récupérées dans les tests manuels, ce qui permet au final d'améliorer considérablement la couverture d'application permise par le

4. Ici, l'auteur parle principalement des tests réalisés avec la suite Sélénium

crawleur simple.

Dans un premier temps, il est nécessaire d'analyser les tests manuels existants afin d'en extraire la connaissance de l'application. Pour ce faire, les interactions réalisées avec le DOM de chaque test sont utilisées pour établir un « *state transition graph* » (STG)temporaire. Le STG temporaire est soumis à Crawljax comme base pour explorer l'application et découvrir les chemins alternatifs.

Les assertions manuelles sont utilisées pour détecter les fonctionnalités importantes et récurrentes à tester. Chaque test portant sur un élément générique du template (bannière, etc.) sera répété sur tous les états de l'application. En même temps, des mécanismes sont mis en place pour éviter de générer trop d'assertions inutilement, qui rendraient la tâche de maintenir les tests trop délicate.

L'outil proposé est disponible [5] sous licence open source. Testilizer est présenté sous la forme d'un plug-in Crawljax. Encore une fois, on remarque que Crawljax est utilisée comme base solide pour faciliter la suite d'un travail reposant sur le besoin de crawler un site. Testilizer permet d'améliorer la génération de tests à partir d'une suite déjà établie, sans empêcher de personnaliser par la suite les tests existants.

3.7 JAEK

L'article de Pellegrino et al. (2015) remet en cause l'efficacité des solutions de crawl actuelles. Il se positionne en nouveauté dans la méthodologie de crawl d'applications 2.0 grâce à une méthode de redéfinition des fonctions de base de JavaScript.

En redéfinissant la fonction *addEvenListener*, nerf de la guerre pour détecter les éléments

5. `https://github.com/saltlab/Testilizer`

déclenchant un changement d'état, Jaek ne risque donc pas de manquer de lien vers les nouveaux états de l'application. Là où tous les autres crawlers tâtonnaient en essayant de repérer de façon arbitraire les événements de chaque application JavaScript, Jaek impose une nouvelle méthodologie puissante.

Cette analyse d'exécution de l'application est utilisée pour rendre le crawleur attentif à tous les comportements de l'application.

Pendant la période de crawl, Jaek maintient un graphe de navigation complet, lui permettant de prendre en compte l'état complet de l'application à chaque changement d'état, afin de mieux décider vers quelles directions crawler.

Il s'avère qu'en comparant les résultats aux autres outils existants, Jaek repousse les limites de l'état de l'art des crawlers actuel en termes de couverture d'états (2502 états découverts sur 13 applications, contre 142 pour Crawljax, pour ne citer que lui).

Cet outil est disponible [6] sous licence GPL 3. Jaek demande à être implémenté sous forme de programme pour être fonctionnel. En effet on constate qu'en soi, Jaek n'est pas capable de gérer l'exécution de tests d'aucune manière que ce soit. Il va de soi que l'intégration continue n'est pas mise en place étant donné qu'aucun test n'est utilisable via Jaek. L'outil ne correspond donc pas vraiment à nos attentes en matière de test logiciel, mais reste une base extrêmement prometteuse en matière de crawl d'application web 2.0.

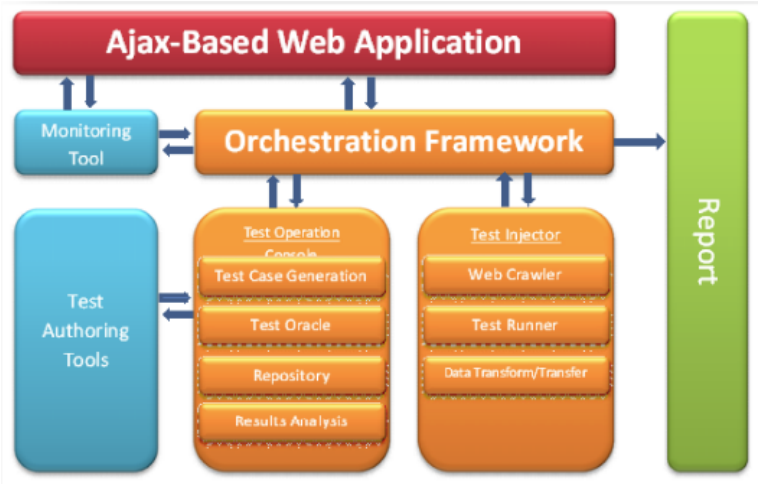
3.8 ORCHESTRATION

L'article de Deyab et Atan (2015) se concentre sur la création d'un modèle théorique d'orchestration du processus complet de test automatisé d'application web.

6. `https://github.com/ConstantinT/JAEk`

L'auteur décrit de façon complète les concepts impliqués dans son étude : crawler, générateur, exécuteur, réducteur et interpréteur de tests, capteur. Autant de termes spécifiques au domaine du test automatisé. À partir de là, l'auteur définit un cadre de travail pour organiser au mieux l'ensemble de tous ces concepts. La figue 3.3, tirée de leur article, résume visuellement l'organisation des concepts et le fonctionnement des modules entre eux.

Figure 3.5 – Modèle théorique d'orchestration de test automatisé d'application web



Cet article est le seul de tout le corpus à réellement prendre en compte les besoins des testeurs de A à Z et à imaginer un flux de travail, théorique, complet de travail.

L'article ne livre aucune implémentation de leur modèle théorique, bien qu'une série d'impressions écrans suggèrent que les auteurs l'aient réalisé dans un cadre privé. Selon la théorie, il semble tout à fait possible de rédiger des tests personnalisés et de les exécuter automatiquement. L'intégration continue n'est évoquée à aucun point, ce qui laisse, en plus d'une implémentation open source, la voie ouverte à l'amélioration du modèle théorique présenté.

L'outil semble correspondre à une majorité de nos critères établis, mais l'absence de livrable rend l'évaluation complète impossible, et la description du mode d'utilisation délicate. Si la partie théorique est dans la lignée de notre objectif et semble convaincante et efficace, nous

considérerons malgré tout que l'outil n'est pas en accord complet avec le besoin établi dans le cadre de cet article. En effet, le caractère utilisable du livrable reste un point important dans nos critères.

3.9 RÉSUMÉ ET ANALYSE

Article	Logiciel	Personnalisation	Intégration Continue
WebMate	oui	non	oui
Crawljax	oui	oui	non
Webmole	oui	non	non
Regression testing	non	non	non
Automated System Testing	oui	oui	non
Leveraging existing tests	oui	oui	non
Jaek	oui	non	non
Orchestration	non	oui	non

3.10 DISCUSSION

Littérature existante Les articles analysés sont triés par ordre de publication : du plus ancien au plus récent. On constate une réelle évolution dans le domaine au fil de la lecture. On passe d'un sujet relativement neuf et simplement compréhensible à des articles scientifiques longs, poussés, et de plus en plus complexes. Des termes spécifiques font surface, une réelle littérature se met en place, et les logiciels sont de plus en plus performants.

On constate que Crawljax est clairement l'outil le plus avancé dans son domaine et le plus souple, grâce à son système de plug-in. Pour autant, depuis son apparition, Crawjax ne semble

pas avoir particulièrement percé dans le domaine du test d'applications de l'industrie. On constate également que Crawljax n'a reçu aucune mise à jour depuis fin 2015, et a récemment été surpassé en termes de couverture d'application par le nouveau crawler Jaek. Codé en Python, ce dernier semble être le meilleur candidat comme base de crawl pour un outil répondant aux besoins de l'industrie. Afin de s'assurer de ce statut de leader, il serait judicieux de faire passer à Jaek le protocole de test de Hallé et al. (2014). En effet, leur protocole de test est transposable aux crawlers modernes, sans limitation de technologie ou d'implémentation.

Le modèle théorique proposé dans l'article de Deyab et Atan (2015) semble, quant à lui, être prometteur et proposer un environnement assez complet pour être utile et adoptable par l'industrie. Bien que quelques concepts définis par les auteurs semblent trop détaillés et risquent de ne pas laisser la liberté nécessaire aux acteurs de l'implémentation du modèle, la base théorique reste solide et réfléchie.

Il serait judicieux d'adapter et de généraliser leur modèle afin de permettre un travail d'implémentation plus libre et plus souple. Des concepts non évoqués dans l'article pourraient également être ajoutés, par exemple l'intégration continue.

Pistes Si le domaine du test d'application web par crawler a beaucoup évolué en cinq ans, on peut toutefois constater que l'état de l'art n'est pas rendu à un stade d'exploitation industrielle. En effet, les logiciels proposés ne semblent pas avoir percé hors du monde académique. En ce décembre 2016, la recherche "crawljax", logiciel le plus populaire du domaine étudié, sur Google retourne moins de 7 500 résultats Google (2016).

Selon nous, la prochaine étape dans le domaine est l'élaboration d'un modèle théorique global basé sur le travail de Deyab et Atan (2015). En effet, le modèle actuel peut être améliorable sur plusieurs points :

1. Le modèle actuel semble entrer dans de nombreux détails d'implémentation. Ces détails risquent d'aller à l'encontre du développement d'une solution efficace et optimisée, deux besoins au coeur du monde non académique. Or, il nous semble plus important de permettre, par exemple, la réutilisation de code existant afin de limiter l'effet "réinventer la roue", mais aussi de repartir sur une base de code existant déjà éprouvé. Ainsi, le modèle à proposer pourrait se permettre d'offrir un plus grand niveau d'abstraction sur les concepts mis en jeu.
2. L'Intégration continue, standard du développement d'applications (toutes technologies confondues) depuis plusieurs années. Il est aujourd'hui impensable pour un outil ayant pour objectif d'être utilisé dans un contexte d'impératifs de production de ne pas implémenter les standards définissant le milieu. L'IC permettra à l'outil de s'ajouter aux méthodes de travail déjà existantes, et rendrait l'exécution des tests entièrement automatique.

Une fois le modèle bien établi, il conviendrait d'en effectuer une implémentation open source moderne. La mention Open Source peut ne pas sembler importante ou pertinente, mais pour de nombreuses entreprises, l'accès au code source est un gage de qualité, et surtout d'indépendance. En effet, même si le projet n'est pas activement maintenu, les utilisateurs auront la garantie de pouvoir maintenir leur propre version, voire de la modifier.

Le caractère moderne de l'implémentation sera défini par deux axes majeurs :

1. Des choix technologiques et logiciels modernes ;
2. Une base d'outils à jour et maintenus.

Pour ce faire, la base du crawler Jaek semble être la plus adaptée. L'outil est développé en Python, langage bénéficiant d'une grande communauté et d'une librairie standard conséquente.

Ce critère permettra d'avoir un outil fonctionnel relativement rapidement, et d'éviter l'effet "réinventer la roue". De plus, Jaek est aujourd'hui le crawler le plus efficace en termes de couverture d'application, et qui semble être le plus mis à jour [7]. Toutefois, le caractère prometteur de Jaek sera à tempérer en fonction des résultats des tests réalisés grâce à l'outil de Hallé et al. (2014), qui pourraient mettre en avant des points à améliorer dans le logiciel ou des technologies à prendre en compte.

7. Le dernier commit de Crawljax remonte au 15 décembre 2015 et l'outil compte environ 70 remontées de bug ouvertes.

Deuxième partie

Modèle théorique

CHAPITRE 4

ORCHESTRATION FRAMEWORK : UNE BASE DE TRAVAIL

La publication de Deyab et Atan (2015) est utilisée comme base théorique pour le développement de notre nouveau travail. Toutefois, comme nous allons le voir, nous n'avons pas gardé l'intégralité de leur approche. Ce chapitre présente notre approche de leur modèle, afin d'introduire notre implémentation finale.

4.1 L'HÉRITAGE

Quels sont les concepts pertinents à garder ?

Crawler web Le crawler web est une brique essentielle sur laquelle va reposer notre projet. C'est lui qui va nous permettre de récupérer l'ensemble du site, et donc d'avoir la matière même sur laquelle faire travailler nos tests. Le crawler devrait être capable de parcourir à la fois des sites web classiques, mais également des applications web 2.0 et modernes, afin de permettre de tester toute la diversité du web actuel. Il est donc primordial que nous puissions récupérer l'état du DOM des pages après chaque interaction du crawler avec le JavaScript.

Notre crawler devra être un composant totalement indépendant du reste de l'application,

afin de respecter l'architecture modulaire voulue, permettant à Cowtest de ne pas être lié de façon définitive au crawler implémenté. Nous ne nous concentrerons toutefois pas sur la programmation du crawler web, domaine complexe et spécialisé s'il en est. Cowtest sera capable, dans un premier temps, de travailler avec une liste d'url en entrée. En plus de respecter l'architecture modulaire visée, cela nous permettra de nous concentrer sur la logique métier du test running, tout en laissant le champ ouvert à l'intégration par après d'un crawler moderne et approprié.

Il est important de noter que nous devrons programmer un moyen standardisé de communiquer entre le crawler et Cowtest. En effet, l'objectif est de pouvoir interchanger le crawler à l'avenir. Non pas d'une façon complètement transparente "plug-and-play", il faudra obligatoirement faire des adaptations du code, mais nous souhaitons garder cette étape d'adaptation la plus simple possible. Il s'agira donc très probablement d'écrire une interface logicielle à respecter pour adapter le fonctionnement du crawler à Cowtest.

Test Runner Une suite de test n'est rien sans un test runner approprié. Le test runner est chargé de faire le lien entre les tests écrits et le code à tester. C'est également le test runner qui va être chargé de gérer le parallélisme des tests, la gestion mémoire, la remontée des diagnostics de tests, etc.

Dans notre cas, le test runner est la librairie qui va lancer les tests à proprement parler. Cette librairie sera choisie par l'utilisateur, et retournera ses informations à Cowtest. Étant donné que le choix de la librairie est laissé à l'utilisateur, il est impératif de développer un moyen standard de communiquer. C'est par le biais du concept des "connecteurs" que nous arriverons à éclaircir cette communication, comme nous allons le voir dans les paragraphes suivants.

Test Oracle En test logiciel, l'oracle est la partie logicielle chargée de faire la comparaison et d'établir si un test est réussi ou non. Le concept est un incontournable pour tout test logiciel. Dans notre cas, le test Oracle sera inclus dans la librairie de test de l'utilisateur final. Cette partie est abstraite de notre usage direct par le connecteur, qui se charge de faire remonter les statistiques du lancement des tests à Cowtest. Ainsi, on peut constater que notre approche se fait à un niveau d'abstraction un peu plus élevé que le modèle théorique de Deyab et Atan (2015).

4.2 LES PARTIES LAISSÉES DE CÔTÉ

Monitoring tool La présence d'un outil de monitoring n'est, selon notre approche, pas indispensable, et sort du champ d'expertise de notre outil. En effet, monitorer le site testé demande une architecture totalement différente de l'application prévue, pour cette seule fonctionnalité. La complexité du développement, et de la maintenance s'en verrait augmentée drastiquement. De plus, le concept d'un outil d'orchestration de test frontend sur l'ensemble d'un site reste un apport au domaine de la recherche, et de l'industrie. Par contre, il existe déjà des outils de monitoring spécialisés extrêmement performants, qui dépassent tout ce que nous serions capables de faire. Nous avons donc décidé de faire l'impasse sur la brique "Monitoring Tool", qui nous semblait hors champ avec nos objectifs, et non atteignable d'un point de vue qualité de développement.

Test Authoring Tools Dans notre cas, la partie édition et rédaction des tests incombe à l'utilisateur final. En effet, la possibilité de lancer des tests écrits avec n'importe quelle librairie fait que nous ne pouvons prédire ni le langage ni l'environnement privilégié par l'utilisateur. Ainsi, Cowtest exploitera les tests spécifiés, sans avoir plus ample connaissance

de leur nature.

Nous laissons le soin à l'utilisateur de choisir son environnement de développement de tests.

CHAPITRE 5

COWTEST : LES APPORTS

5.1 NOUVEAU CONCEPT : LES CONNECTEURS

Afin d'être capable de lancer n'importe quelle librairie de test sur les données récupérées par le crawler, il est nécessaire d'avoir une partie logicielle flexible, qui va s'adapter à chaque librairie. C'est le rôle du connecteur. C'est un pont entre l'application Cowtest et la librairie de test choisie par l'utilisateur. Le connecteur est chargé de transformer les informations de l'un, en un langage compréhensible pour l'autre, et donc de retourner des données normées à Cowtest.

Il est nécessaire d'établir un langage normé pour la sortie d'informations des librairies de test utilisées. De même, le message retourné à Cowtest doit être fixe et défini à l'avance. Une documentation spécifique et une interface logicielle à respecter seront nécessaires, afin que le programmeur du connecteur puisse savoir comment construire ce pont.

Cela ne pose pas de problème de sécurité dans notre cas, car Cowtest est destiné à être lancé sur la machine du testeur. Si notre solution aurait été hébergée sur une infrastructure dédiée en SAAS » [1], il aurait été vital d'analyser le code fourni par les testeurs afin d'éviter toute

1. Software As A Service, plus communément appelé application cloud

opération malicieuse.

5.2 ABSTRACTION DU MODÈLE THÉORIQUE

Sans vouloir simplifier à outrance, l'objectif de notre modèle théorique est de rajouter une certaine abstraction sur l'existant, et de le rapprocher d'un modèle plus propice à une intégration fonctionnelle. En effet, l'objectif final est de construire une application fonctionnelle, pas d'établir un modèle théorique parfait, mais inutilisable. Nos briques théoriques seront donc pensées en perspective d'une implémentation réaliste.

Selon notre approche, le modèle de Deyab et Atan (2015) est extrêmement complet, mais également très complexe. En effet, intégrer le log serveur et l'outil d'édition de tests, par exemple, dénote une volonté de "tout faire", qui donne une réelle force au modèle théorique, mais rend l'implémentation du modèle extrêmement complexe, et probablement irréaliste.

5.3 DÉPENDANCES SIMPLES, MODULARITÉ FORTE

Notre objectif est de construire quelques modules indépendants, ouverts et fortement documentés, puis de les faire fonctionner ensemble dans une implémentation de notre modèle conceptuel. La finalité est de multiple :

1. Pouvoir programmer et mettre à jour de façon totalement indépendante chaque module ;
2. Permettre une meilleure réutilisation et maintenance du code ;
3. Permettre aux utilisateurs d'utiliser les modules pour les implémenter dans leurs solutions. On pourrait imaginer utiliser un crawler différent, voire récupérer le fichier Sitemap » [2] en lieu et place d'un crawler, voire utiliser l'entrée standard UNIX ;

2. Fichier XML situé à la racine de nombreux sites, permettant de lister l'ensemble des pages du site. En

4. Pouvoir arriver à une solution fonctionnelle plus rapidement, en utilisant dans un premier temps des modules préexistants, avant de coder les briques spécifiques à notre modèle théorique.

Cette modularité permettra de respecter clairement et simplement le modèle initial, puisque chaque module pourra être représenté directement par une brique théorique de Cowtest.

Nous nous efforcerons de garder la plupart du temps un couplage par données, où les modules s'échangeront des informations par le biais d'arguments de type simples. Certains modules devront toutefois avoir un couplage externe, puisque récupérant des données externes dans des fichiers de stockage. À noter que les inconvénients du couplage fort sont gardés au minimum, puisque chaque module peut seulement lire les données du module précédent, et doit écrire dans un nouveau fichier. Cela permet d'éviter d'avoir à gérer des cas de conflits d'écriture, en particulier.

5.4 SCHÉMATISATION DU MODÈLE THÉORIQUE

La figure suivante rappelle le modèle théorique de Deyab et Atan (2015) :

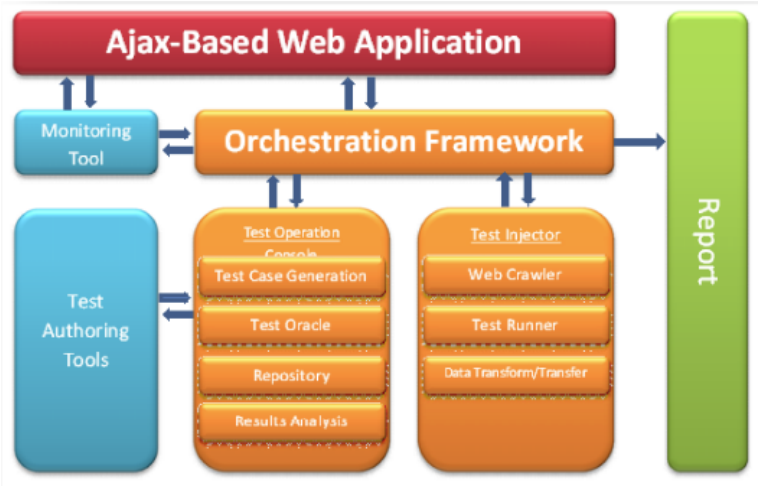
En comparaison, suis notre modèle de pensée :

Nous constatons plusieurs points essentiels à la comparaison de ces deux modèles :

1. Notre approche se veut plus haut niveau, en allant chercher un plus petit nombre de composants théoriques, et en se cantonnant à une spécialité plus restreinte.
2. Sans que tout soit parfait, nous avons essayé de garder un sens unique de communication des composants. Ainsi, les dépendances devraient rester les plus logiques possible, et les composants les plus indépendants les uns des autres. Dans le modèle

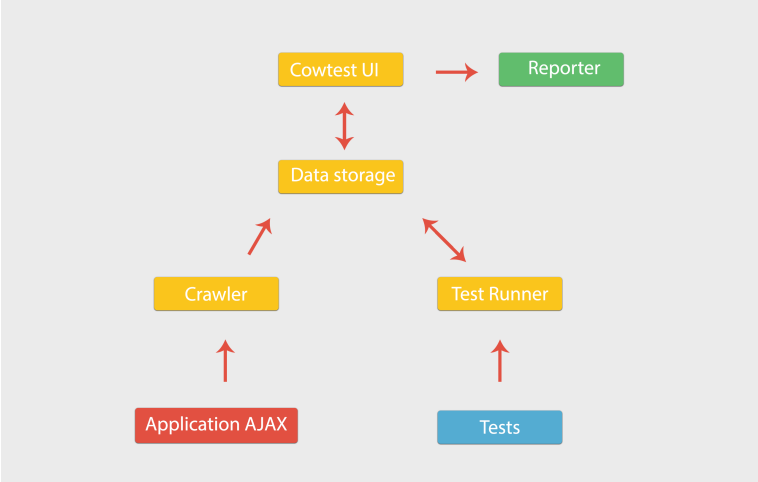
général avec des visées de référencement sur les moteurs de recherche.

Figure 5.1 – Modèle théorique de l'Orchestration Framework utilisé en base de notre travail



original de Deyab et Atan (2015), à l'inverse, tous les modules (excepté le reporter) communiquent à double sens.

Figure 5.2 – Mise en évidence de l'architecture logicielle



Troisième partie

Implémentation

CHAPITRE 6

CONCEPTION

6.1 CAS D'UTILISATION

Pour justifier l'utilisabilité de notre solution dans le domaine de l'industrie, nous allons énumérer quelques cas d'utilisation Cowtest pertinents.

6.1.1 ANALYSE SEO

Un cas plutôt simple, qui pourrait potentiellement avoir des investissements importants est celui de l'analyse de référencement. En effet, les référenceurs font aujourd'hui des audits plus ou moins manuels des sites sur lesquels ils travaillent. Cowtest pourrait permettre de gagner un temps considérable. En effet, à l'heure actuelle, il est difficile pour les professionnels référenceurs de s'assurer que leurs normes sont respectées au travers du site au complet. En général, des pages "templates" sont définies, et seule une occurrence de chaque page est testée (un produit, un article de blog, etc.). Cowtest pourrait permettre d'avoir la certitude qu'aucun cas particulier n'échappe au test. De plus, il est en général reconnu que le domaine du référencement est très subjectif, et que chaque expert peut avoir des stratégies très différentes. Cowtest permettant d'écrire n'importe quel test fonctionnel, il s'avère très adapté aux

différentes stratégies des nombreux référenceurs sur le marché.

6.1.2 AUDIT D'ACCESSIBILITÉ

Utiliser un site avec un lecteur d'écran est une expérience que tout créateur de contenu web devrait s'astreindre à essayer un jour. L'utilisation d'un très grand nombre de sites populaire devient un calvaire en seulement quelques minutes. Cowtest pourrait permettre ici aussi de lancer une batterie de tests précis sur les pages d'un site internet afin de s'assurer que le balisage du contenu est efficace et correct. On pourrait également imaginer remonter des informations comme les tags alternatifs manquants d'images, certaines pratiques non optimales, voire, par extension, l'ensemble des points purement techniques des normes d'accessibilité.

6.1.3 ANALYSE DE QUALITÉ DE CONTENU

Il existe de nombreux sites qui offrent du contenu d'auteurs variés. Dans le cas d'un site restreint, cela ne pose pas vraiment de problème, mais pour un portail, un blogue collaboratif, ou un forum de grande ampleur, vérifier manuellement tous les nouveaux contenus pourrait poser problème. On pourrait tout à fait imaginer brancher Cowtest sur l'application web décrite pour s'assurer de certaines variables au travers de tout le site. Par exemple, la qualité de la langue, le respect de normes d'écriture, le hors sujet, etc. Rappelons que l'écriture des tests est toujours laissée à la charge de l'utilisateur.

6.1.4 ANALYSE DE NORMES DE DESIGN GRAPHIQUE

Le redesign d'une application legacy peut être une tâche longue et délicate. Ainsi, grâce à Cowtest, il est possible de lancer des tests graphiques sur toutes les pages de l'application.

À l'heure actuelle, il est délicat de suivre la propagation des modifications graphiques dans une application large existante. Ainsi, plusieurs versions d'un même logo peuvent parfois se côtoyer au sein d'une application, ou encore des teintes de couleurs légèrement différentes. Parfois, on constate un mélange de plusieurs kits d'icônes, ou encore un mélange de plusieurs typographies héritées d'anciennes versions. Autant de points qui seraient testables simplement avec Cowtest.

6.2 DU MODÈLE THÉORIQUE À LA CONCEPTION LOGICIELLE

Pour la partie conceptuelle du logiciel, nous avons opté pour une approche plutôt simple. Inspirés du fonctionnement modulaire du gestionnaire de packages de NodeJS »[1], Cowtest est découpé en plusieurs modules métiers simples, eux-mêmes utilisant d'autres modules.

Les concepts de Crawler, Reporter et Test Runner sont donc devenus autant de modules éponymes, ayant comme règle principale de fonctionner de façon totalement indépendante, et êtres exploitables par une API claire et définie à l'avance.

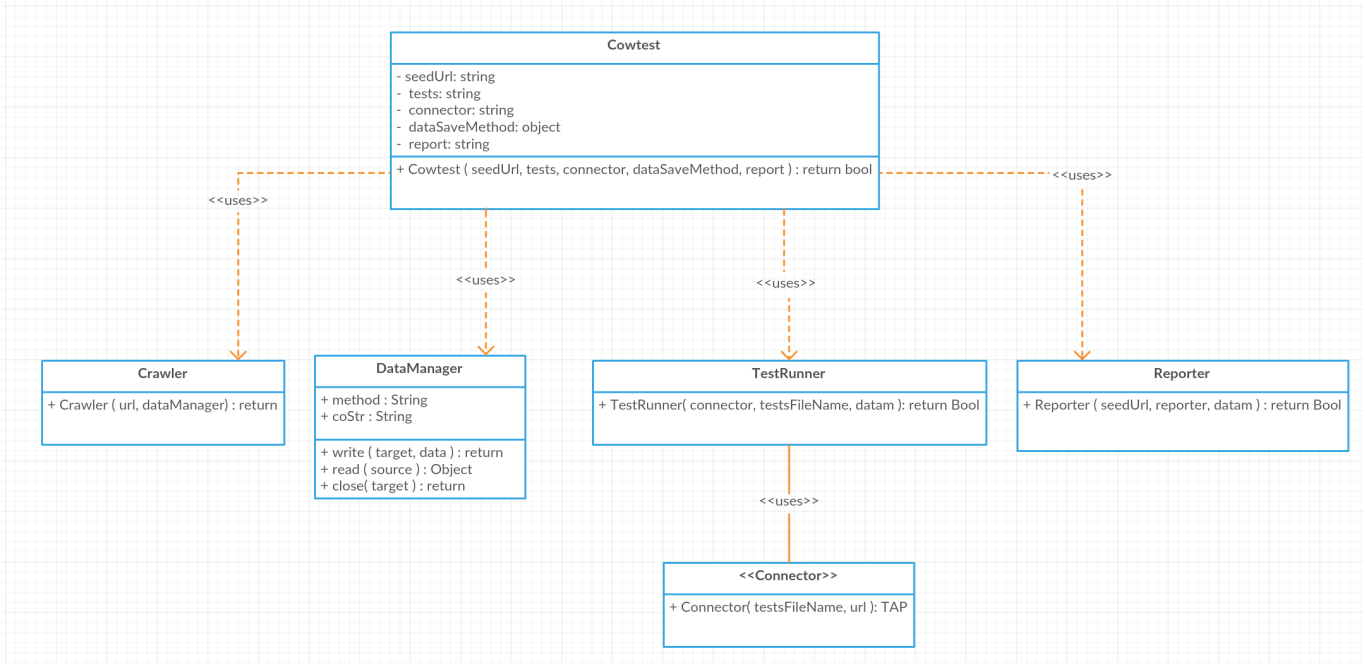
6.2.1 DIAGRAMME DE CLASSES

Notre logiciel sera divisé en six grandes classes. Chaque classe sera programmée sous forme de module entièrement indépendant.

Nous constatons facilement la ressemblance entre le diagramme de classes de Cowtest et le schéma du modèle théorique exposé dans la section précédente.

1. Npm, pour Node Package Manager

Figure 6.1 – Diagramme de classes de Cowtest



Cowtest Coeur du programme, c'est ce module qui va être en charge de gérer, orchestrer et récupérer tous les autres modules de notre logiciel. Purement utilitaire, ce module ne réalisera pas de réel calcul ou algorithme métier. Ce module demande un certain nombre d'informations indispensables au lancement du logiciel : l'URL de base à visiter, le nom de fichiers de tests à lancer, le nom du connecteur (sous forme de chaîne de caractère) ou bien une fonction. En effet, nous utilisons les capacités de programmation fonctionnelle de JavaScript afin de passer le code du connecteur directement dans l'argument). Il est également nécessaire de préciser sous quelle forme les données doivent être sauvegardées, ainsi que le reporter à utiliser pour faire remonter les données. Ici aussi, ce peut être le nom d'un reporter par défaut, ou une fonction contenant directement le code du reporter custom.

Crawler Prenant en argument l'URL de base à visiter, ainsi qu'une instance du DataManager (qui va, comme nous allons le voir, nous permettre d'abstraire la sauvegarde de données).

Actuellement, le crawler utilise un module externe (développé par nos soins, mais qui ne fait pas partie intégrante du projet Cowtest), permettant un crawl d'application 1.0 ainsi qu'une récupération du sitemap XML. Côté code, tout sera pensé de façon à pouvoir interchanger simplement le crawler.

DataManager Module purement utilitaire permettant d'abstraire la sauvegarde de données, il permet de gérer de façon transparente l'enregistrement, que ce soit dans des fichiers textes ou une base de données. Le module prend deux arguments : la méthode de sauvegarde, qui doit être une des chaînes de caractères proposées par défaut, et la chaîne de connexion. À noter que dans le cas de la sauvegarde sur fichiers, la chaîne de connexion deviendra de façon transparente le nom du fichier de sauvegarde.

TestRunner Coeur de notre logiciel, c'est ce module qui va permettre de gérer le lancement des tests de l'utilisateur. Récupérant en argument le DataManager, le connecteur et le nom du fichier de tests originellement spécifiés au module "cowtest", c'est ici que chaque test est lancé via le mécanisme du connecteur (voir le paragraphe suivant).

Connector Lien entre Cowtest et les tests écrits par l'utilisateur, c'est le connecteur qui va transformer ses arguments d'entrée (fichier de test et URL) en un output compréhensible par Cowtest. Les URL à tester sont passées sous forme de variables d'environnement aux tests, exécutés dans autant de sous-processus. A priori, l'output du connecteur se fera en JSON.

Reporter Point final de l'exécution de Cowtest, le reporter permet simplement d'avoir un retour des résultats d'exécution de Cowtest. À noter que le reporter ne retourne pas d'information au module Cowtest, il est indépendant, et en charge de retourner les informations

sous la forme désirée par l'utilisateur. Par défaut, il est capable de retourner des informations sous deux formes intégrées à Cowtest : log console, ou output HTML. Il est également possible de transmettre une fonction (encore une fois, par le biais de la programmation fonctionnelle), et de programmer un reporting custom.

6.2.2 DESIGNS PATTERNS IMPLÉMENTÉS

Connecteurs : strategy Le connecteur est en soit une implémentation du design pattern strategy. En effet, Cowtest dispose d'une série de différents algorithmes à lancer selon les préférences entrées par l'utilisateur. Il permettra, selon le choix de l'utilisateur, de lancer l'algorithme de connexion adapté à la librairie de test. Plusieurs algorithmes seront livrés avec l'application, permettant de connecter quelques librairies considérées comme standard dans Cowtest, tout en laissant la possibilité à l'utilisateur final de passer en argument son propre algorithme. À noter que ce passage de fonction en argument devra utiliser des concepts de programmation fonctionnelle.

Data Manager : singleton La gestion des données se fait par le biais d'une couche d'abstraction, le Data Manager. Purement utilitaire, cette classe est un mécanisme nous permettant de changer de support de sauvegarde de données simplement en spécifiant un argument au lancement de Cowtest. Cela permettra de livrer des méthodes de sauvegarde de données par défaut (fichier, base de données, etc.), tout en permettant de garder une architecture ouverte à l'implémentation d'autres modes de traitement.

CHAPITRE 7

CHOIX TECHNOLOGIQUES

7.1 LANGAGE

Plusieurs facteurs sont entrés en ligne de mire pour le choix du langage :

1. L'objectif du projet : certains langages ont des "prédispositions" pour certaines tâches, et permettent un travail plus efficace.
2. La communauté du langage : notre objectif n'est pas de produire un outil puis de l'abandonner. Dans une optique de développement au long terme, un langage moderne avec une forte communauté et des outils de développement solides est primordial.
3. Le langage doit permettre une livraison de l'outil simple et efficace pour l'utilisation par les utilisateurs finaux.

L'objectif final étant l'analyse et le test de la partie front-end de sites internet, il paraît logique d'utiliser la technologie leader du domaine : JavaScript. Depuis l'arrivée de Node.js, JavaScript a connu un réel renouveau, et est aujourd'hui un langage incontournable, autant côté serveur que côté client.

Le gestionnaire de modules de Node.js, npm, est un modèle de simplicité, et permet l'installa-

tion d'un script complet en une seule ligne de commande. Pas de compilation manuelle ni de librairie conséquente à installer par un autre canal de distribution.

S'il faut un dernier argument concernant la maturité du langage et de l'environnement NodeJS, il est notable que la NASA à publiquement déclaré utiliser NodeJS et NPM pour programmer leurs combinaisons spatialesNode.js Foundation (2017).

Selon notre approche, JavaScript semble être le langage le plus approprié à notre problème, et semble également être l'outil qui permettra à l'outil de toucher plus de développeurs, et par la même, d'être testé par le plus grand nombre.

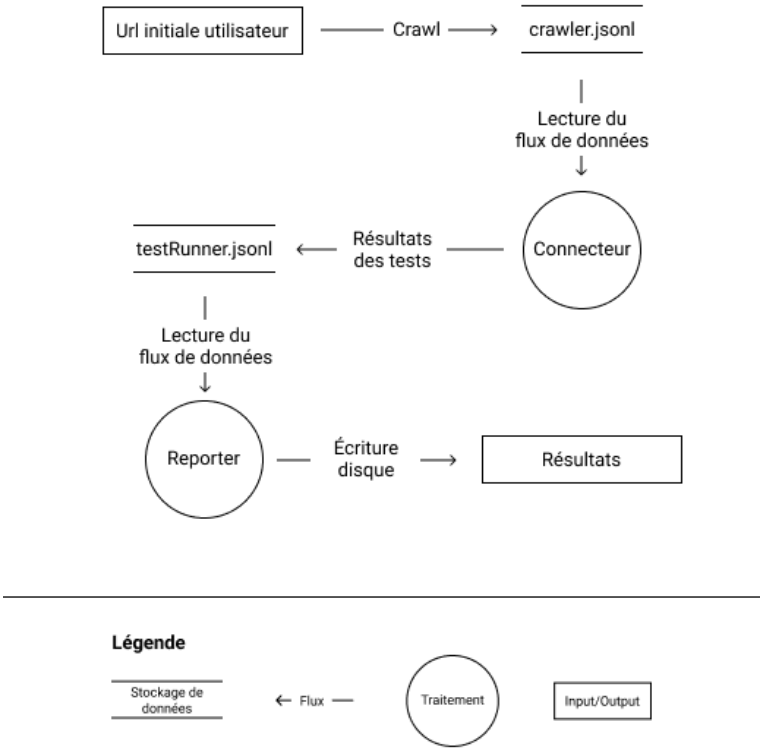
7.2 DONNÉES ET STOCKAGE

Les données récoltées tout au long du processus de traitement sont cruciales. Dans l'ordre d'acquisition :

1. L'URL de départ
2. Toutes les URL issues du crawl, basé sur l'URL de départ
3. Un ensemble de données décrivant les tests lancés et leur statut (succès ou échec).

Le diagramme de flux de données suivant (figure 8.1) permet de comprendre visuellement le chemin que parcourent les données au sein de l'application. On remarque clairement que les données sont écrites et lues sur le disque à deux reprises au cours du cycle de vie de l'application. Bien que cela puisse paraître peu optimisé, voire redondant, cela permet de libérer la mémoire au fil de l'utilisation, et donc d'avoir au final la capacité de traiter un grand nombre d'informations sans saturer la RAM. Cela ouvre l'utilisation de Cowtestà des sites de grande taille.

Figure 7.1 – Diagramme de flux de données de Cowtest



Le format retenu pour l'enregistrement des données est le JSONL [1]. Le format JSONL consiste en une suite de JSON valide, séparés par un retour à la ligne UNIX. Le fichier au complet n'est pas un JSON valide en soi, mais cela permet de traiter les données en flux, ce qui donne une souplesse et une rapidité d'exécution supérieure à n'importe quelle méthode de stockage bloquante.

La limite du stockage de données est donc actuellement égale aux limitations du système de fichiers. Dans les faits, un site de taille petite à moyenne (notre estimation est d'environ 10 000 pages) fonctionne bien. Dans un second temps, il pourrait être pertinent d'implémenter une base de données plus robuste, type MongoDB (qui fonctionne particulièrement bien avec des objets JSON). Cela permettrait de pousser la robustesse de Cowtest un niveau plus loin, et

1. `http://jsonlines.org/`

de supporter des sites bien plus larges.

Il est également possible de demander une sortie en JSON. Sans aucune configuration, Cowtest renvoie les résultats de son travail dans la sortie standard sous forme d'un objet JSON. Simplement réutilisable, il est possible de le stocker dans un fichier, ou de le passe en pipe à un autre processus.

7.2.1 TAP

Afin de communiquer les résultats de tests, l'utilisation du standard TAP a été retenue. TAP est un format de description des résultats d'une suite de tests. Défini par TAP format contributors (2017), le format général est le suivant :

```
TAP version 13
1..N
ok 1 Description # Directive
# Diagnostic
  ---
  message: 'Failure␣message'
  severity: fail
  data:
    got:
      - 1
      - 3
      - 2
    expect:
      - 1
      - 2
```

```
                                        - 3
                                      ...
                                    ok 47 Description
                                    ok 48 Description
                                    more tests....
```

Dans les faits, une sortie TAP valide pourrait être la suivante :

```
TAP version 13
1..4
ok 1 - Input file opened
not ok 2 - First line of the input valid
  ---
  message: 'First␣line␣invalid'
  severity: fail
  data:
    got: 'Flirble'
    expect: 'Fnible'
  ...
ok 3 - Read the rest of the file
not ok 4 - Summarized correctly # TODO Not written yet
  ---
  message: "Can't␣make␣summary␣yet"
  severity: todo
  ...
```

7.3 MODULARITÉ DU CODE

L'objectif de développement à court terme étant avant tout d'avoir un outil fonctionnel, nous avons mis l'accent sur l'utilisation de librairies connues et reconnues, afin d'éviter de réinventer la roue et de perdre de l'énergie de développement sur des problèmes non spécifiques à notre recherche.

Cette façon d'aborder le développement de Cowtest permet également de respecter nos exigences de modularité. Cowtest sera finalement le ciment entre différentes briques métier discutées dans la partie théorique.

7.3.1 MODULES UTILISÉS

Sans faire une liste exhaustive de chaque module utilisé, les dépendances importantes de Cowtest sont les suivantes :

1. p-queue [2] : Ce module permet de gérer la concurrence des promises JavaScript. Nous l'avons utilisé pour gérer une file d'attente de requêtes web afin de ne pas surcharger les serveurs ciblés. En effet, un trop grand nombre de requêtes simultané provoquait généralement un bannissement de notre connexion pour un court délai, et faisait échouer le processus.
2. tap-parser [3] : Module permettant de parser directement la sortie des librairies de test au format standard TAP [4].
3. jsonl-file [5] : Module minimaliste réalisé par nos soins, permettant d'ajouter une couche

2. `https://www.npmjs.com/package/p-queue`
3. `https://www.npmjs.com/package/tap-parser`
4. `https://testanything.org/`
5. `https://www.npmjs.com/package/jsonl-file`

d'abstraction sur la lecture et l'écriture en flux de fichiers jsonl.

4. dandy-crawl[6] : Crawler simple et efficace que nous avons développé en parallèle à notre recherche, permettant de crawler un site web 1.0 (par URL). Par défaut, le crawler parcourt les liens du site pour récupérer toutes les pages. Il est également capable de croiser ces données avec la liste d'URL récupérées depuis le sitemap, pour une couverture maximale.

7.4 DIFFICULTÉS RENCONTRÉES

Quelques étapes ont été notablement plus complexes à mettre en place que les autres. Par exemple :

7.4.1 PROMESSES JAVASCRIPT ET TRAITEMENT ASYNCHRONE

Cowtest fait un usage intensif des promesses JavaScript. Les promesses sont un mécanisme implémenté depuis la version ES6 de JavaScript, permettant de faciliter l'écriture et la gestion d'opérations non bloquantes. Elles permettent d'éviter l'écriture de callbacks parfois laborieux, et de récupérer tous les résultats des différentes tâches au même endroit.

La quantité de promesses lancées en simultané est notablement élevée dans notre cas. En effet, chaque promesse représente une instance de la librairie de test de l'utilisateur final, multiplié par le nombre de pages à tester.

Dans notre implémentation, nous avons fait le choix de ménager la consommation, en étendant le traitement sur un temps plus long. En effet, lancer toutes les promesses d'un seul coup permet de gagner légèrement en rapidité, mais a pour effet de charger le serveur testé d'un

6. `https://www.npmjs.com/package/dandy-crawl`

très grand nombre de requêtes d'un coup. Limiter le nombre de promesses concurrentes à une valeur donnée permet de ne jamais assiéger l'application, tout en ajoutant un temps de traitement raisonnable. À noter que la valeur maximale de concurrence reste à déterminer. Pour l'instant, la valeur peut être spécifiée par l'utilisateur, permettant donc de faire différents essais afin de trouver la valeur la plus efficace pour le cas d'utilisation.

7.4.2 IMPLÉMENTATION DES CONNECTEURS ET LANCEMENT DES SOUS-PROCESSUS

D'un point de vue complexité de programmation, permettre la connexion avec n'importe quelle librairie de test était un postulat complexe et osé. Le concept de connecteur nous a permis de rendre cette idée réalisable et soutenable.

En effet, à l'heure actuelle, n'importe quelle librairie de test renvoyant ses résultats en respectant la norme TAP peut être parsée et renvoyée à Cowtest.

Dans les faits, la programmation d'un connecteur est relativement rapide et simple (les connecteurs inclus dans Cowtest sont un très bon modèle d'exemple). L'idée générale est d'écrire un module node exportant une promesse ES6. Cette promesse doit lancer la librairie de test, parser sa sortie TAP grâce au module “”tap-parser“” (présenté précédemment), et renvoyer l'objet JSON associé.

CHAPITRE 8

L'OUTIL

8.1 INTERFACE

Cowtest fonctionne uniquement en ligne de commande à l'heure actuelle. Ce format est pratique pour plusieurs raisons :

1. Rapidité de développement : développer une interface en ligne de commande est évidemment un choix stratégique vis-à-vis du temps de développement. Développer une interface graphique complète demanderait un temps conséquent. Bien évidemment, dans le cadre d'une utilisation élargie par des utilisateurs externes, une interface utilisateur graphique ergonomique et moderne serait un point important, mais au stade actuel du développement, il est tout à fait acceptable que les utilisateurs aient à utiliser leur terminal pour profiter de Cowtest.
2. Interopérabilité : un outil en ligne de commande est très facilement intégrable à n'importe quel script. En effet, Cowtest est lancé par la commande node, et peut, selon la configuration, afficher ses résultats sur la sortie standard, dans un format personnalisable par l'utilisateur. Utiliser Cowtest dans un flux de travail devient dès lors beaucoup plus simple.

8.2 EXEMPLE D'UTILISATION

Afin de mieux comprendre le fonctionnement de Cowtest, nous allons décrire un cas complet d'utilisation, de l'installation, à l'interprétation du rapport de bug.

8.2.1 INSTALLATION

L'utilisation de Cowtest se fait programmatiquement, en intégrant l'outil à un script JavaScript de l'utilisateur. Il est nécessaire d'installer Cowtest dans le répertoire du script final, par le biais de la commande "npm install". Le code suivant est contenu dans un fichier nommé "index.js".

```
import Cowtest from 'cowtest';


const seedUrl = "http://localhost/"; // URL du site a tester
const tests = `${__dirname}/tests.js`; // Emplacement du fichier de tests
const connector = "ava"; // Le connecteur a utiliser pour lancer le fichier de
    tests
const report = "html"; // La methode a utiliser pour faire remonter le
   diagnostic final
const dataSaveMethod = { method: 'jsonl', coStr:`${__dirname}/data.jsonl` } //
    Methode de sauvegarde de donnees


Cowtest({
  seedUrl,
  tests,
  connector,
  dataSaveMethod,
```

```
  report
});
```

Le script ci dessus peut être divisé en trois blocs :

1. Import de Cowtest dans le script ;
2. Déclaration des variables de personnalisation du lancement de Cowtest ;
3. Exécution du logiciel, avec passage en argument des variables personnalisées.

8.2.2 ÉCRITURE D'UN TEST

Dans cet exemple, nous allons utiliser la librairie de test "ava" [1] (c'est pourquoi dans le script initial, nous avons spécifié le connecteur "ava"). À noter qu'une grande partie de la syntaxe du code qui suit est propre à cette librairie (comme la fonction "test" par exemple). À noter qu'il aurait tout à fait été possible d'utiliser un autre langage, comme Python, pour ce script, JavaScript est un choix de l'utilisateur final.

Le code qui suit, contenu dans le fichier "test.js" va nous permettre de tester deux points distincts. La première fonction "test" vérifiera la validité W3C du code HTML, la seconde, vérifie l'absence de tout message de niveau warning ou error dans la console JavaScript.

```
// La fonction ‘‘test’’ est une fonction publique de la librairie de test
    utilise dans cet exemple.
// Nous faisons usage du mot clef ‘‘async’’, disponible dans la derniere
    version de JavaScript afin de beneficier de clarifier la syntaxe de la
    partie asynchrone.
test('Should␣be␣validated␣by␣the␣W3C', async (t) => {
```

1. https://github.com/avajs/ava

```
function validate(validateUrl) {
  return new Promise(function(resolve, reject) {
    // Nous utilisons un module open source se chargeant de requeter le site
        du W3C et retourner le resultat en JSON.
    w3cjs.validate({
      file: validateUrl,
      output: 'json',
      callback: (err, res) => {
        resolve(res);
      }
    });
  });
}

// Le mot clef await permet de rendre l'operation suivante bloquante. A
    noter que cela fonctionne parce que nous avons encapsule notre fonction
    ``validate'' dans une Promise ES6.
const res = await validate(URL);

// Si le W3C renvoie des erreurs, on fait echouer le test, en renvoyant les
    erreurs
if (res.messages.length > 0) {
  t.fail(JSON.stringify(res.messages, null, null));
} else {
// Sinon le test est valide
  t.pass();
}
```

```
});

test('Console␣shouldn\'t␣output␣error␣or␣warning␣logs', async (t) => {
  // Nous utilisons un moteur webkit sans tete, controlable programmatiquement
       : Nightmare.
  const nightmare = Nightmare();
  const errors = [];
  const warnings = [];

  // Le mot clef await permet de rendre l'execution du code a venir bloquante.
       A noter que cela fonctionne parce que Nightmare fonctionne nativement
      avec des Promises ES6.
  await nightmare
    .on('console', (type, message) => {
      // On enregistre chaque message d'erreur
      if (type==='error')
        errors.push(message);
      // On enregistre chaque message de warning
      if (type==='warn')
        warnings.push(message);
      })
    .goto(URL)
    .end();

  if (errors.length > 0 || errors.length > 0) {
    // Si nous avons enregistre des messages d'erreur ou de warning, on fait
        echouer le test, en renvoyant les erreurs
```

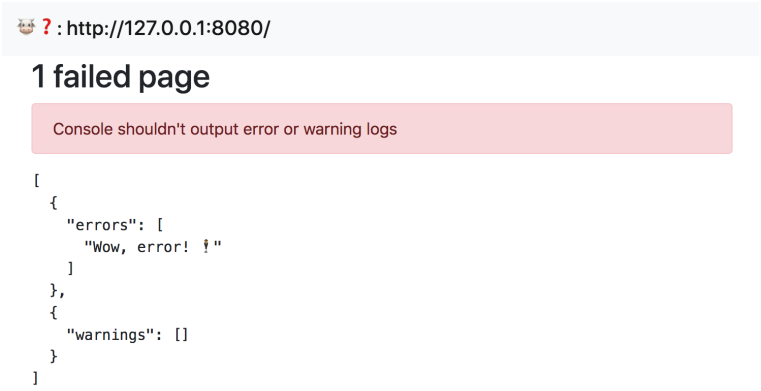
```
    t.fail(JSON.stringify([ {errors}, {warnings} ], null, null));
  } else {
    // Sinon le test est valide
    t.pass();
  }
});
```

Pour le bien de l'exemple, nous utilisons un site statique simple de quatre pages, hébergé localement. Le site contient un code HTML5 valide, aucune erreur ne devrait ressortir sur ce point, mais le JavaScript renvoie volontairement (pour le bien de l'exemple) une erreur dans la console.

8.2.3 EXÉCUTION ET RÉCUPÉRATION DES RÉSULTATS

Avec la commande "node index.js", Cowtest est lancé. Après un temps d'attente grandement variable (de quelques secondes à plusieurs minutes, selon la taille du site), le navigateur s'ouvre automatiquement, afin de nous présenter le diagnostic posé par notre logiciel.

Figure 8.1 – Impression écran du diagnostic web de Cowtest



Comme nous le voyons, Cowtest fait seulement remonter les erreurs détectées. Ainsi, le site d'exemple passe avec succès le test du HTML valide, mais fait défaut au test du log console JavaScript.

8.3 FONCTIONNALITÉS NON IMPLÉMENTÉES

Bien que le développement de Cowtest ait été une préoccupation importante de notre recherche et ait été optimisé grâce à l'utilisation de librairies et de modules préexistants, tout le modèle théorique n'a pas pu être implémenté. Ainsi, les fonctionnalités suivantes n'ont pas pu être mises en place :

8.3.1 CRAWLER

Le crawler tout indiqué pour constituer une base solide à Cowtest était Jaek. Robuste, à la pointe de la recherche, dans un langage moderne et bien supporté. Hélas, toutes nos tentatives d'intégration à notre outil ont été soldées par des échecs. L'installation des dépendances est déjà une épreuve en soi. En effet, le crawler est basé sur une version obsolète de pyQT, qui est complexe à installer, et ne correspond plus à la documentation actuelle. Après de nombreuses heures d'essai, nous avons donc décidé de ne pas intégrer Jaek à notre solution, puisque sa complexité d'installation se refléterait forcément sur la complexité d'installation de notre solution.

Gardant l'idée d'intégrer par la suite un crawler efficace, nous avons décidé de construire une architecture la plus souple et évolutive possible. Ainsi, dans sa première version Cowtest est livré avec un crawler codé par nos soins. Nommé "Dandy-crawl", ce crawler permet de parcourir un site web classique (sans AJAX), et renvoyer toutes les URL récupérées. À

noter que le crawler parse également le sitemap du site, afin de s'assurer d'avoir la meilleure couverture possible, simplement. Cela permet de fonctionner avec la plupart des sites, et de tester Cowtest dans un environnement réaliste. À noter qu'en interne, notre logiciel travaille avec un flux d'URL fourni par le crawler. Il sera donc très simple par la suite d'adapter cette partie pour prendre un flux de n'importe quelle source, que ce soit un pipe de processus, un crawler indépendant, ou un fichier système.

8.3.2 AJAX

Comme une conséquence du point précédent, l'intégration de l'AJAX à également été un point délicat dans le développement de notre application. En effet, à partir du moment où aucun crawler ne nous permettant de récupérer simplement chaque état AJAX d'une application, il était délicat de lancer des tests sur autre chose que le state initial de chaque page. Ainsi, avant de songer à lancer des tests sur l'intégralité des states d'une application, il convient d'être capable de crawler et de récupérer de façon normée l'application AJAX.

8.4 FUTUR DU DÉVELOPPEMENT

Dès l'initiation du projet Cowtest, il était clair que la direction favorisée était celle de l'open source. En effet, le choix d'utiliser un langage moderne et communautaire couplé à une licence open source est selon nous un pas important dans l'objectif de ne pas laisser le logiciel mourir avec la fin de notre période de recherche.

Au-delà de l'intérêt dans le domaine académique, nous sommes convaincus que Cowtest trouvera preneur dans le domaine professionnel non académique. Ainsi, laisser la liberté aux utilisateurs de lire et modifier le code et l'implémentation du logiciel permettra, au

minimum, de laisser la chance à Cowtest d'être utilisé, au mieux, la possibilité de poursuivre son développement au sein d'une communauté d'utilisateurs, et de grandir pour devenir un outil encore plus adapté à son marché.

8.4.1 FEUILLE DE ROUTE

Par ordre de priorité, les fonctionnalités développées seront :

1. Établir une façon définie et structurée de lancer des tests sur un DOM fonctionnel interprétant JavaScript ;
2. Intégrer un crawler compatible AJAX ;
3. Livrer des séries de tests d'exemple préconfigurés et utilisables simplement.

Nous prévoyons également la mise en place d'une documentation en ligne à jour et d'un site pour communiquer sur le projet.

Cowtest est en ligne sur l'hébergeur GitHub, à l'adresse : `https://github.com/tmos/cowtest/`

CHAPITRE 9

CONCLUSION : AU-DELÀ DE COWTEST, PRISE DE RECUL ET OUVERTURE

9.1 CHEMIN PARCOURU

Ainsi, nous avons dans un premier temps analysé la littérature existante. Après avoir constaté une réelle évolution dans les travaux présentés, nous avons toutefois délimité certaines pistes de travail ouvertes, permettant à notre travail de se situer en perspective du travail déjà réalisé.

Nous avons ensuite mis en place un modèle théorique, basé sur Deyab et Atan (2015), nous avons poussé noter réflexion pour avancer sur la base de leur travail. Nous avons apporté un nouveau concept, le connecteur, mais aussi avons ajouté une couche d'abstraction sur le modèle de pensée.

Finalement, nous avons décrit notre implémentation, de la conception logicielle à un exemple d'utilisation de Cowtest, en passant par la justification de nos choix de langage, de gestion des données, le tout sans cacher les difficultés que nous avons rencontré tout au long de notre travail.

9.2 COWTEST : LIMITES ET VISIONS

L'intégration de plusieurs points ont été plus complexes que prévu :

9.2.1 L'INTÉGRATION D'UN CRAWLER

Un des points qui a présenté un niveau de complexité réellement plus élevé a été l'utilisation et l'intégration d'un crawler. En effet, le seul crawler présentant a priori permettant une intégration programmatique était Jaek. Il s'est avéré à l'usage que ce dernier était basé sur un framework Webkit contenu dans une ancienne version de QT. Le décalage temporaire entre le QT ciblé, la documentation disponible, et le manque de documentation d'installation de Jaek à fait qu'il nous a été impossible de réussir à exploiter le logiciel dans l'implémentation de Cowtest.

Pour palier ce problème, Cowtestest tout de même capable de fonctionner grâce à deux mécanismes différents :

1. Récupération du sitemap : si spécifié, Cowtestva chercher le fichier sitemap.xml situé à la racine du site. Ce fichier, au départ créé pour des besoins de référencement auprès des moteurs de recherche, liste l'ensemble des URL mises en avant pour le site consulté. Il est donc une base sûre des URL pertinentes à visiter pour notre logiciel. À noter que, bien que considéré comme une bonne pratique presque incontournable, ce fichier n'est pas obligatoire.
2. Crawl du site non AJAX. Nous avons en effet développé un crawler minimaliste, capable de récupérer l'ensemble des URL d'un site. Il est important de spécifier que le crawler fonctionne sans aucune interprétation du DOM ou du JavaScript. Il nous permet donc de récupérer les URL d'un site classique, permettant donc à Cowtestd'être

fonctionnel et utile à tout un pan du web. La plupart des sites e-commerce par exemple reposent grandement, pour des raisons marketing et référencement, sur une architecture sans AJAX. Ainsi, même si le modèle théorique reste optimal, Cowtestest capable de fonctionner dans un milieu professionnel.

9.2.2 LA PERFORMANCE

Bien que facteur non limitant dans notre projet, il a tout de même été nécessaire de porter un peu plus d'attention que prévu sur ce point. En effet, la charge de travail de l'application peut monter en flèche très rapidement selon la taille de l'application testée. En effet, pour un jeu de 20 tests, lancés sur un site de 400 pages, le nombre de tests à lancer est déjà de 8000. Pour peu que la librairie de tests utilise un navigateur headless (phantomjs, chrome headless, Firefox headless, etc.), ou doive faire appel à des mécanismes externes, la consommation peut très rapidement augmenter en flèche. À noter que la gestion de la performance pure des tests n'est pas de notre ressort. En effet, l'utilisateur pouvant écrire ses tests dans n'importe quels langage ou librairie, la gestion de la mémoire et de la consommation de la ressource lors de l'exécution des tests est hors de notre contrôle.

9.2.3 AJAX

Dans la conception de départ de l'outil Cowtest, AJAX était central dans notre approche. Il l'est bien évidemment toujours dans la partie théorique, mais au fur et à mesure de notre implémentation, il a été de plus en plus clair que ce serait la partie la plus conséquente à réaliser. Dans la version livrée de Cowtest, la navigation n'est possible qu'au sein d'un site classique "1.0", où la cle unique d'identification des pages reste l'URL. Ainsi, le paradigme des applications web 2.0 et modernes n'est pas pris en charge. Notre expérience de développement

nous a cela dit donné quelques pistes de réflexion sur une direction générale de prise en charge de la navigation AJAX, comme nous allons le voir par la suite.

9.3 TRAITEMENT AUTOMATISÉ D'UNE APPLICATION AJAX : DOMAINE COMPLEXE ET PROBLÈME OUVERT

La prise en charge de l'AJAX dans une application web est un problème académique connu, ouvert, et complexe. En effet, au-delà de la simple compréhension technique du mode de fonctionnement d'une application AJAX, au-delà de la détection des éléments dynamiques (point pourtant complexe et principal difficulté de tout crawler AJAX), il s'agit de mettre sur pied une toute nouvelle façon de penser l'application traitée.

9.3.1 ID UNIQUE DE STATE

Comme vu plus tôt dans ce mémoire, dans l'ancien paradigme de traitement des applications web, le contenu était le HTML, et l'identifiant unique, l'URL. Avec les applications web 2.0, le contenu est devenu le DOM, et l'identifiant unique pose toujours problème. En effet, l'URL ne suffit plus, cas pour dévoiler un contenu spécifique, il faut, non seulement la base de l'URL, mais également une suite d'actions (clic, hover, drag, drop, etc.) réalisée sur un ou plusieurs éléments du DOM spécifique.

9.3.2 FORMAT NORMÉ

Il s'agirait donc dans un premier temps d'établir un format de notation fixe pour décrire les différentes actuelleions réalisables sur le DOM.

Sélecteur du DOM : Xpath Le format de ciblage des éléments du DOM xpath est un format existant et éprouvé. En effet, Xpath est une norme du W3C permettant une sélection précise d'un unique nœud d'un document XML. Étendu dans les dernières normes au DOM des pages web non valides XHTML, c'est le format de choix pour sélectionner un élément unique dans une page. À noter que les sélecteurs CSS pourraient également fonctionner, mais ils ne permettent pas de traverser l'arbre du DOM dans le sens enfant-parent. A priori, Xpath est à l'heure actuelle la solution la plus souple et performante.

Actions sur le DOM La liste des actions détectables sur le DOM est extrêmement longue et précise. Tous les évènements possèdent déjà un code précis et un nom unique. Quelques évènements qui pourraient nous intéresser seraient par exemple :

1. click
2. contextmenu
3. dblclick
4. mousedown
5. mouseenter
6. mouseleave
7. mousemove
8. mouseout

Vers un standard L'idée serait de définir une syntaxe de langage déclaratif permettant de reproduire de façon unique une suite d'interactions précises sur le DOM d'une URL. Ainsi, sur une version donnée de l'application web, avec une URL de base et la suite d'opérations décrites, l'utilisateur serait sûr d'arriver sur le même état du DOM. Les tests utilisateurs de Cowtestpourraient alors être lancés à partir de cet état du DOM.

9.3.3 UN CHAMP DE RECHERCHE POSSIBLE

Le traitement et la prise en charge d'une application AJAX sont possibles, et même déjà faits dans certaines recherches. Pour autant, il semble évident que cette prise en charge est un domaine notablement complexe et délicat. Il nous semble nécessaire de prendre du recul sur ce processus.

En effet, établir un processus optimisé et normé de communication avec les applications AJAX apporterait une réelle amélioration au domaine. Cela permettrait d'avoir une base solide et commune aux futures recherches, tout en évitant à chaque nouveau projet de réinventer la roue. De plus, avoir un langage de communication stable et fiable, permettrait de créer des projets interopérables, chose importante pour pouvoir partager les avancées réalisées.

9.4 RETOURS SUR LES OBJETIFS

Comme prévu au début de ce mémoire, nous souhaitions faire un retour sur nos objectifs initiaux :

Réfléchir à une façon efficace et appropriée de tester un site complet Nous avons en effet rapidement déterminé une façon de faire fonctionner ensemble le crawler et la librairie de test, deux points centraux de notre stratégie de test. Ces deux concepts nous semblent toujours les plus appropriés au problème initial de tester l'intégralité d'un site web.

Déterminer un modèle théorique et une architecture fiable Sur la base approuvée du travail de Deyab et Atan (2015), nous avons développé notre propre modèle théorique, permettant ainsi de bénéficier d'une base solide, tout en apportant un nombre conséquent de modifications

et d'appropriations propres à nos objectifs.

Développer une implémentation fonctionnelle respectant au plus près notre modèle théorique Nous avons réalisé en cours de développement qu'implémenter la totalité de notre modèle théorique serait une tâche extrêmement complexe, et que cela demanderai une force de développement conséquente. Nous avons donc du faire des choix d'implémentation, afin de garder un certain réalisme dans le projet.

D'une façon générale, nous pensons avoir convenablement respecté nos objectifs initiaux, bien que l'implémentation ai été revue avec un peu moins d'ambition, cela a tout de même permis d'établir un bon exemple, validant ainsi l'utilité de notre idée de départ. Il nous semble au final tout à fait légitime d'assurer que la majorité de nos objectifs ont été atteints.

BIBLIOGRAPHIE

Dallmeier, V., M. Burger, T. Orth, et A. Zeller. 2012. « Webmate : A tool for testing web 2.0 applications ». In *Proceedings of the Workshop on JavaScript Tools, JSTools'12*, p. 11–15.

Deyab, H. H. et R. B. Atan. 2015. « Orchestration framework for automated ajax-based web application testing ». In *2015 9th Malaysian Software Engineering Conference, MySEC 2015*, p. 1–6.

ECMA International. 2017. ECMAScript 2017 language specification. `https://www.ecma-international.org/publications/standards/Ecma-262.htm`.

Google. 2016. Crawljax - google search. `https://encrypted.google.com/search?hl=en&q=crawljax`.

Hallé, S., G. L. Breton, F. Maronnaud, A. B. Massé, et S. Gaboury. 2014. « Exhaustive exploration of ajax web applications with selective jumping ». In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, p. 243–252.

IEEE. 1990. IEEE standard glossary of software engineering terminology. Rapport no. Std 610.12-1990, IEEE.

Le Breton, G., F. Maronnaud, et S. Hallé. 2013. « Automated exploration and analysis of ajax web applications with webmole ». In *WWW 2013 Companion - Proceedings of the 22nd International Conference on World Wide Web*, p. 245–248.

Mesbah, A., E. Bozdag, et A. v. Deursen. 2008. « Crawling ajax by inferring user interface state changes ». In *Proceedings of the 2008 Eighth International Conference on Web Engineering*. Coll. « ICWE '08 », p. 122–134, Washington, DC, USA. IEEE Computer Society.

Mesbah, A. et A. van Deursen. 2009. « Invariant-based automatic testing of ajax user interfaces ». In *Proceedings of the 31st International Conference on Software Engineering*. Coll. « ICSE '09 », p. 210–220, Washington, DC, USA. IEEE Computer Society.

Mesbah, A., A. van Deursen, et D. Roest. 2012. « Invariant-based automatic testing of modern web applications », *IEEE Transactions on Software Engineering*, vol. 38, no. 1, p. 35–53.

Milani Fard, A., M. Mirzaaghaei, et A. Mesbah. 2014. « Leveraging existing tests in automated test generation for web applications ». In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*. Coll. « ASE '14 », p. 67–78, New York, NY, USA. ACM.

Mozilla Developer Network. 2018. Guide ajax. `https://developer.mozilla.org/fr/docs/Web/Guide/AJAX`.

Node.js Foundation. 2017. Node.js helps NASA keep astronauts safe and data accessible. `https://foundation.nodejs.org/wp-content/uploads/sites/50/2017/09/Node_CaseStudy_Nasa_FNL.pdf`.

Pellegrino, G., C. Tschürtz, E. Bodden, et C. Rossow. 2015. *JÄk : Using dynamic analysis to crawl and test modern web applications*. T. 9404, p. 295–316.

Raina, S. et A. P. Agarwal. 2013. « An automated tool for regression testing in web applications », *SIGSOFT Softw. Eng. Notes*, vol. 38, no. 4, p. 1–4.

Sébastien, A. 2013. L'intégration continue et ses outils. http ://igm.univ-mlv.fr/ dr/X-POSE2012/Integration

Tanida, H., M. R. Prasad, S. P. Rajan, et M. Fujita. 2013. *Automated System Testing of Dynamic Web Applications*. Coll. « Communications in Computer and Information Science ». T. 303, p. 181–196.

TAP format contributors. 2017. Test anything protocol. `https://testanything.org/`.

Thummalapenta, S., K. V. Lakshmi, S. Sinha, N. Sinha, et S. Chandra. 2013. « Guided test generation for web applications », *ICSE : International Conference on Software Engineering*, p. 162–171.

World Wide Web Consortium. 2004. W3C DOM specification. `https://www.w3.org/DOM/DOMTR`.

———. 2015. W3C HTTP specification. `https://www.w3.org/standards/techs/http#w3c_all`.

———. 2017a. W3C CSS specification. `https://www.w3.org/Style/CSS/specs.en.html`.

———. 2017b. W3C HTML specification. `https://www.w3.org/TR/html52/`.